PHYSICAL SIMULATION OF AN EMBEDDED SURFACE MESH INVOLVING DEFORMATION AND FRACTURE

A Thesis

by

BILLY RUSSELL CLACK

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2012

Major Subject: Computer Science

PHYSICAL SIMULATION OF AN EMBEDDED SURFACE MESH INVOLVING DEFORMATION AND FRACTURE

A Thesis

by

BILLY RUSSELL CLACK

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	John Keyser
Committee Members,	Ergun Akleman
	Jinxiang Chai
Head of Department,	Duncan M. (Hank) Walker

May 2012

Major Subject: Computer Science

ABSTRACT

Physical Simulation of an Embedded Surface Mesh Involving Deformation and Fracture. (March 2012)

Billy Russell Clack, B.M., Stephen F. Austin State University

Chair of Advisory Committee: Dr. John Keyser

Simulating virtual objects which can deform or break apart within their environments is now common in state-of-the-art virtual simulations such as video games or surgery simulations. Real-time performance requires a physical model which provides an approximation to the true solution for fast computations but at the same time provides enough believability of the simulation to the user. Recent research in object deformation and fracture has revolved around embedding portions of the simulation for graphical display inside a much simpler physical domain which is invisible to the user. Embedding complex geometry in a simpler domain allows for very complex effects to occur in a much more robust and computationally efficient manner. This thesis explores a novel method to efficiently embed a high-resolution surface mesh inside a coarse tetrahedral physical mesh for the purposes of interactive simulation and display. A technique to display interior regions as solid geometry without explicitly re-meshing the graphical mesh during fracture has been explored and developed. Keeping the graphical mesh static in memory during simulation allows the geometry to be off-loaded to the GPU while shaders can be utilized to only display portions of the geometry which are locally contained within the physical mesh. Recent advances in GPU technology have also been exploited in order to provide an increase in visual fidelity and help achieve the illusion that the virtual object itself is breaking apart in a physically plausible manner.

To My Parents

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

FIGURE Page

FIGURE Page

CHAPTER I

INTRODUCTION

Real-time environments require that the underlying physical representation of a scene be combined with the graphical representation in a way that allows for real-time performance. One method of achieving interactivity is to separate what the user sees from the underlying physical systems at play. When complex effects such as the breaking-up of objects are considered, the physical models underlying the simulation become complex and oftentimes computationally restrictive. Recent advances in embedded methods allow for an evolution of a coarse physical model to update a fine resolution graphical model. While many of these methods produce plausible results, the mesh oftentimes must be re-meshed during fracture in order to represent the internal boundaries. Furthermore, many physical simulations still require the graphical mesh in order to perform physical processes such as collision detection. It is advantageous to keep the entire graphical mesh which the user sees static in memory (no addition/deletion/merging of triangles) to allow efficient computation and display of the mesh.

In this thesis I explore methods which help speed up current deformation/fracture simulations while allowing the graphical model design process to be completely agnostic of the physical representation used in the simulation. I will explore if it is possible to use an embedding method to divorce entirely the physical computations from the graphical representation of the mesh. This work will show how complex geometry can be efficiently embedded inside a low-resolution physical mesh and then be simulated over time. The physical model may deform due to elastic stress and

---

The journal model is *IEEE Transactions on Automatic Control*.

strain, and subsequently may break apart. Since the physical mesh is agnostic of the graphical mesh, the physical simulation is not slowed down due to a high graphical polygon count. Since the mapping from high-to-low degrees of freedom will undoubtedly have approximation errors associated with it, an investigation into how close the embedding mapping comes to the best theoretical possible solution will be performed. The methods developed will allow for the graphical mesh to remain static in GPU memory while giving the illusion that it is being deformed/broken apart to the user. Since no re-meshing is required, a method to close off the gaps associated with interior regions will be investigated by using texture mapping in a preprocessing step. Because the goal of this research is to be create a believable simulation, GPU functionality will be exploited to help provide detail in areas of the mesh which may appear to be too flat due to the linear nature of the physical tetrahedral mesh.

To summarize, the main contributions of this work are the following:

1. Completely separate the graphical representation of the simulation mesh from the physical representation in a virtual environment.

2. Use the physical representation to perform physical simulations involving deformation and fracture.

3. Show how the boundaries of the physical mesh may be used to display the graphical mesh directly by using modern graphics processing unit (GPU) functionality without re-meshing the input assets.

4. Experimentally determine the error associated with the collision detection/response system.

5. Utilize modern GPU features in combination with displacement mapping to display portions of the mesh which contain no embedded geometry but should

be filled (interior portions).

A. Motivation

Physically based deformation of complex geometry for real-time scenarios has increasingly been researched in the past, with applications from video games to virtual surgery simulations. The continuing necessity for a higher state of visual fidelity has allowed for complex, visually stunning simulations which appear to react as they would in the real world. The development of efficient algorithms for deforming materials in a virtual world has allowed for amazing simulations taking place simultaneously in a user's virtual environment. Common materials which may be seen in current big-budget titles include deformable and breakable solids, cloth, and even fluid simulations, all at real-time frame rates.

While the computational power of modern platforms continues to increase, the visual fidelity of the physics simulations will naturally increase as well. The sheer performance power of future systems will provide increased complexity and believability. With the added increase in memory, digital assets will become increasingly complex as well. There is still a strong need for algorithms which are both computationally efficient and allow asset artists to focus on the content (3D models in this case) without worrying about how the complexity of the model will adversely affect the performance of the physical simulations.

One method to couple the graphical model with a coarse physical model is to embed the graphical mesh within a finite element mesh composed of tetrahedrons as in Figure 1. A problem arrises when discontinuities occur since the graphical mesh emulates a solid object but is only represented as the boundary of the object. Remeshing the graphical model to close off holes and discontinuities also requires complex

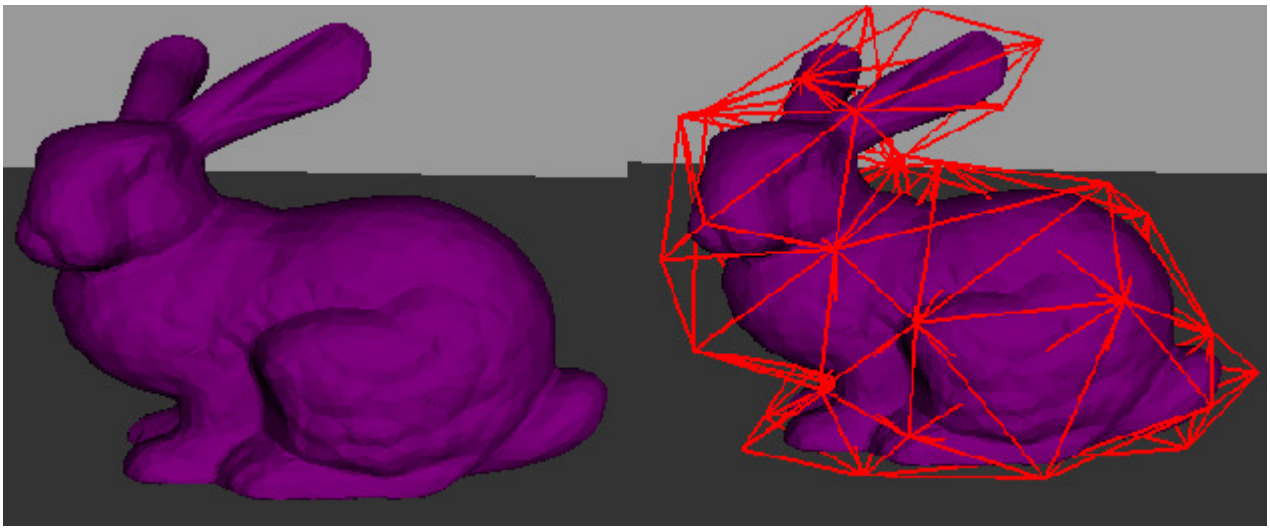
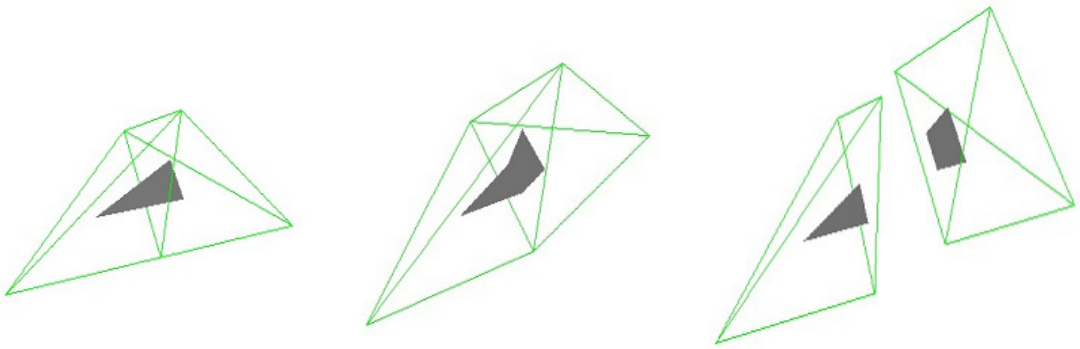


Fig. 1. The bunny enclosed in a finite element tetrahedral mesh. The mesh is first generated as a control triangle mesh and then tetrahedralized.

geometric calculations to occur during simulation time. Furthermore, modern graphical processing units (GPUs) are very efficient in drawing geometry stored locally in the GPU's memory, but constantly transferring geometry information across the CPU-GPU bus is an expensive operation.

In the finite element mesh, tetrahedrons are used as elements and the nodes connecting them form the basis functions for the elements. In order to keep the graphical geometry static in GPU memory, a method to represent the interior portions of a graphical mesh must be developed which does not arise from a re-triangulation step. In order to solve this problem, cracks and breaks that form can be confined to tetrahedral boundaries allowing the initial tetrahedral mesh to not be re-meshed. A mapping between graphical mesh and finite element mesh can be created, and during animation this mapping is evaluated to derive the graphical positions relative to the finite element positions. The physical mesh is a much coarser mesh than the graphical mesh (see Figure 1). The obvious advantage to this approach is that the graphical performance and physical performance are decoupled from each other. The basic

notion of embedding is what inspired this research and is fundamentally connected to the idea of free-form deformation. See Sederberg and Perry [1] for the basis of free-form deformation.



(a) No deformation (b) Deformation (c) Fracture

Fig. 2. An example triangle embedded inside two tetrahedral elements. 2(a) An initial undeformed mesh. A single triangle is embedded inside two tetrahedra. 2(b) One tetrahedron moves, although both tetrahedra are still connected. 2(c) The two tetrahedra become disconnected. The embedded triangle becomes implicitly disconnected as well.

Even if the tetrahedra are used to perform the breaking of an object, the interior boundaries will still appear linear since they are essentially the walls of tetrahedrons (see figure 2) Given that the graphical geometry is separated exactly at the walls of their embedding tetrahedrons, the interior walls must be roughened in order to appear more realistic.

B. Overview of Research

In this work, methods were explored which would allow for the simulation mesh (for physics) and the rendered mesh (for graphics) to be properly divorced during

simulation. The inputs to the simulation are a triangle model for graphical display and a tetrahedral control mesh which encloses the entire graphical model. The high-level overview of the simulation is outlined in the following:

1. Preprocessing stage. During this stage, assets are created and stored for later use during the simulation. This step must only be run once for each graphical model and tetrahedral control mesh. The assets created include additional graphical geometry and collision geometry.

2. Simulation initialization. During this stage a mapping between graphical mesh and physical mesh is created. Triangles of the graphical mesh are assigned tetrahedra based on which ones they need to be embedded within. Display lists are created and geometry is offloaded to the GPU for subsequent display.

3. Simulation run. The simulation processes in discrete time steps. External influences may distort the shape of the meshes which will give rise to internal stresses and strains. If the material fails, a fracture occurs and the physical mesh's connectivity is updated accordingly.

4. Graphical animation. The graphical object is drawn to the screen. The mapping from physical space to graphical space is performed in a shader using precomputed weights. Portions of the physical mesh may have been partially textured during the preprocessing step which could be displayed to the user as well.

A preprocessing step is first performed. The preprocessing system uses as input a triangle mesh and a tetrahedral control mesh. The preprocessing step serves the following main purposes:

1. Create the collision geometry of the object directly from the graphical mesh. The collision geometry itself is a coarse approximation of the "true" geometry

of the mesh.

2. Create graphical geometry which is not part of the input mesh and add this geometry to the graphical mesh's triangle list.

3. Close off interior portions of the graphical mesh along the facial boundaries of the embedding tetrahedrons.

The preprocessing stage contains the most expensive portions of the simulation. This stage does not need to be run every time the simulation begins, but instead it should only be run once during asset generation. Once the assets are created, they can be loaded every time the simulation begins along with the graphical model and the tetrahedral model.

The simulation algorithm itself can be described almost entirely in terms of the physical tetrahedral mesh. Unlike methods which rely on the graphical geometry to perform at least a subset of the physical calculations (such as collision detection), in this work only the invisible coarse portions are used to perform physics calculations. Stresses and strains are computed within the physical mesh using standard physical models adapted from classical mechanics by O'Brien and Hodgins [2]. These stresses and strains represent internal forces occurring inside the physical model. Depending on the material of the model, these forces may exceed the material limit which would induce a fracture plane. Following the work done by Parker and O'Brien [3], instead of re-meshing individual elements that become discontinuous, a fracture plane is snapped to the boundaries of the elements. While this technique produces physically inaccurate results, it still produces physically *believable* results and prevents costly re-meshing. Once the fracture plane is determined, elements may become disconnected. The discontinuity is represented solely by the connectivity structures, which in this case are the nodes of the mesh. Fracture will cause these nodes to split

apart and separate adjoining tetrahedra.

During a simulation the physical mesh is invisible to the user and instead the user sees the results of the mapping to the graphical object. Due to the availability of programmable graphics hardware, the mapping of the graphical mesh from the embedding mapping can be done entirely on the GPU in a shader. A novel technique to break geometry is performed entirely in the GPU as well by exploiting the interpolation functionality inherent in the graphics pipeline.

Collision detection and reaction is another major component of any physically based simulation. While collision detection could be performed using the graphical mesh, it makes much more sense to also divorce the collision model from the actual graphical model as well. In most production environments, artists are constrained to created meshes which conform to a maximum polygon count. The reason for this is so the mesh itself doesn't cause a bottleneck in producing real-time frame rates. While polygon count will always affect the graphical performance of a system, it would be nice to at least decouple the physical portion from the mesh. By utilizing this decoupling, the artist has more freedom to design more complex and polygon-dense meshes. This research aims to develop a collision system which also does not depend on the graphical mesh during the simulation. In this research, a method was designed to measure the best possible collision reaction given the embedding paradigm. This method essentially allows us to compute a lower-bound on the error expected from any collision system which uses the mapping from fine-to-coarse embedding. Experiments are performed which measure how close this system comes to the best possible lower-bound error solution.

The organization of this thesis is as follows. In Chapter 2, background work in the field of physically based simulation as well as fracture simulation will be reviewed. In Chapter 3 the preprocessing phase will be discussed. In Chapter 4 the formulas

relating to deformation and fracture will be discussed as well as how the fracture plane is computed. The approximating fracture plane is primarily inspired by Parker and O'Brien [3]. In Chapter 5, the graphical components of the simulation will be discussed as well as an overview of how tessellation shaders are used to help displace geometry. Chapter 6 will give an overview of the results. This chapter will also contain the results of experiments to determine the error of the collision response of the system as well as how close to the best possible solution the simulation comes. Finally, Chapter 7 will conclude with possible future research as well as a discussion of the pitfalls discovered during the course of this research.

CHAPTER II

BACKGROUND

A. Early Work

Early work in applying physical deformations to computer simulations borrowed very much from classical mechanical engineering literature. These methods often required integrating complex energy functions across the solid. The main pioneering work in applying these energy functions to computer graphics was Terzopoulos et al. [4]. While the results included very simple shapes and the calculation of each frame was by no means quick, their work laid the framework for applying their ideas in future research. Their research was also some of the first to show how to apply general physical models to graphical simulations. A year later, Terzopoulos and Fleicher [5] showed how add permanent plasticity and fracture to their simulations.

Late in the 90's, a few key breakthroughs were made in the realm of time integration methods and deformation/fracture. Baraff and Witkin [6] were able to derive an implicit time integration scheme which allowed for very stiff materials while relaxing the small time step requirement. The method developed in this seminal work was used as the integration scheme of choice for this research. Key work in applying fracture mechanics to computer graphics was performed by O'Brien and Hodgins [2] in 1999. They showed how to properly compute strains and stresses in order to induce discontinuities in the material. While their method was slow and required re-meshing, it laid the foundation for future research in graphical fracture. A few years later, O'Brien et al. [7] extended previous research to include plasticity in the object. Plasticity allows for the simulation of a model which would be permanently deformed if the stresses exceeded a certain threshold (material-dependent).

B. Time Integration Methods

While not the essential focus of this thesis, it is worthwhile to present an overview of typical physically based time integration schemes. In Hauth and Etzmuß' work [8], an overview of time integration steps and an overall architecture was explained. Volino and Magnenat-Thalmann [9] gave a theoretical and empirical comparison of popular integration schemes when applied to a deformable cloth simulation. Baraff and Witkin [6] were the first to show how an implicit integration scheme could be used in physically based modeling, specifically with applications to cloth simulation. The implicit time integration scheme allows for large stable time steps.

C. Free-Form Deformation

Free-form deformation allows geometry to be embedded within a control mesh for an artist to easily manipulate and morph the geometry. The field of free-form deformation has evolved since the mid 1980's. While the earliest work focused on giving a digital artist more freedom to design content and warp shapes, the ideas fundamental to that field have been extended to other applications, including physically based modeling.

Work has been done in free-form deformation which involves embedding geometry into a control mesh. The control mesh may be modified which will, through a mapping function, implicitly modify the embedded mesh. Sederberg and Parry [1] showed one of the first methods of using free form deformation techniques. Others such as Coquillart [10], MacCracken and Joy [11], and Faloutsos et al. [12] extended FFD for various scenarios. Of particular application to physically based systems was Melek and Keyser's work [13] where they simulated a complex physical process by mapping forces arising inside a control mesh to drive the visual simulation of combustible

material. Nesme et al. [14] showed how to derive a mapping to embed elements in a non-homogenous method, which allowed for different material types within the same mesh.

D. Mesh-Based Fracture

Research in fracturing physical objects has been abundant in the last few decades. One of the earliest papers which showed how to fracture objects in a physical environment is by O'Brien and Hodgins [2]. In this paper, they used an explicit finite element method to model forces over the volume of the object to be deformed or fractured. Their work was based off of standard elasticity theory from continuum mechanics literature. O'Brien et al. [7] showed how their algorithm could be used with a plastically deformable object as well. With a modification to the rest shape of the object, ductile fracture which induces a permanent deformation in the object was realized. Smith et al. [15] discretized the volume of the objects into tetrahedra and used point-mass constraints to hold the object together. When forces became to high, the geometry was broken up along the faces of the tetrahedra. They use a fractal subdivision algorithm to make the elements look like shards rather than tetrahedra. Their time integration scheme was based off of constraint methods developed a few years earlier. Müller et al. [16] showed how static analysis could be used to allow fracturing in real time. In their algorithm, they treated the stiff object as a rigid body until contact, at which point they treated the region around a collision as an influence region and fractured that portion of the mesh while leaving the remaining portion of the object fixed in space. They solved for the static equilibrium equation, which relates the internal forces arising in an object to the external forces. Müller et al. [17] were able to factor out rotations from individual finite elements in order to use

linearized strain tensors. Etzmuß et al. [18] showed how Cauchy's linear strain tensor could be applyed to a finite element simulation with large deformations by first factoring out a rotation from each element's stiffness matrix. The results were applyed to a cloth simulation. Müller and Gross [19] fractured a surface mesh embedded in a lower resolution coarse control mesh. Rotations were factored out of each tetrahedron before calculating stress / strain as in Etzmuß et al. [18]. Once rotations were factored out, they could use a linear, non-rotationally invariant Cauchy stress which is computationally cheaper to compute. Unlike the method proposed here, their method required re-meshing the underlying surface mesh in order to simulate the fracturing of the embedded geometry. Muller et al. [20] showed how a mesh represented only by the surface mesh could be fractured. Their algorithm required discretizing space into hexadedra and using the nodes of the hexahedra to close off holes forming between the surface mesh and the internal volume of the object. Their method still required remeshing in the cases of fracture. Bao et al. [21] used the finite element method with respect to thin shell simulations. Their simulation allows for fracture as well as plastic deformations. Parker and O'Brien [3] showed how fracturing could be accomplished in a real production scenario. They used the same fracture algorithm as O'Brien and Hodgins [2], but instead restricted crack propagation to the faces of the tetrahedra. They also took advantage of parallel processing to speed up computations for a real-time scenario. Molino et al. [22] used tetrahedral elements to simulate cutting material. Their method allowed tetrahedra to be partially filled with material during the simulation. During collisions, a triangle boundary for the interior portions of the tetrahedra was created and added to the element boundary list for collision detection and handling. Wicke et al. [23] utilized current research in tetrahedral robustness to create an elastic and plastic simulation which could involve splitting by dynamically re-meshing over the time step intervals to prevent degeneracies. Their algorithm used

both a simulation space model and a rest space model with a mapping from one to the other. Remeshing happens on the rest space mesh, and as the mesh deforms, the rest space mesh could also deform which naturally leads to plastic behavior in the simulation mesh. Sifakis et al. [24] allowed explicit cutting of tetrahedral elements while displaying an embedded high-resolution mesh. Their method is exceptionally well suited to rigid body simulations as well as precise cuttings/incisions. For an overview fracture mechanics, it is recommended to see Anderson [25].

E. Mesh-less Methods

There have been significant advances in the realm of mesh-less deformation and fracture as well. Mesh-less techniques do not keep an explicit topology of the mesh being simulated, but instead usually consist of a discrete set of sampled points within an object which represents the volume or surface of the object. Pauly et al. [26] extended the mesh-less approach and used a dynamic re-sampling approach during the fracture of the object. In their approach, they used an initial volumetric sampling which may be dynamically updated and re-sampled as the object deforms or breaks apart. Müller et al. [27] presented a method to perform shape matching and integrated the equations of motions by defining target shapes and moving the current shape towards its respective target. The target shape was derived from the relationship of the current configuration of the object to the rest configuration of the object. Sifakis et al. [28] gave a method to deform a solid based on a distribution of particles. Hard binding constraints were used to set target positions and soft binding constraints were used to allow force integration and propagation throughout the triangular mesh.

F. Organic Visualization and Surgery Simulation

Mendoza and Laugier [29] showed how to use the finite element method in order to cut through soft tissue. Their method required re-meshing into multiple tetrahedra. Because tetrahedra were used to represent the fracture mesh, a degeneracy such as a single connecting vertex may arise. For muscle visualization, Teran et al. [30] and [31] gave a geometric formulation of deformation using the finite volumetric method. They used tetrahedra to visualize deforming muscles which were allowed to contract and deform. They embeded complex geometry in a coarse tetrahedralized mesh, similar to the methods used in this research. In the case of topologically disconnected geometry which falls into the same tetrahedron, they created tetrahedra which may be coincident but contained distinct geometry. See Nealen et al. [32] for a survey of deformation research.

CHAPTER III

PREPROCESSING

In order to efficiently embed and simulate the graphical mesh at run-time, a preprocessing phase is required. During the preprocessing phase, assets are created such as textures and collision geometry which will be loaded and used during run-time. While the preprocessing phase is the most computationally complex and data-intensive step of the simulation, it must be performed only once during content generation. Subsequent runs of the simulation will simply load the results of the preprocessing phase from disk. The overview of the components of the preprocessing section can be seen in Figure 3.

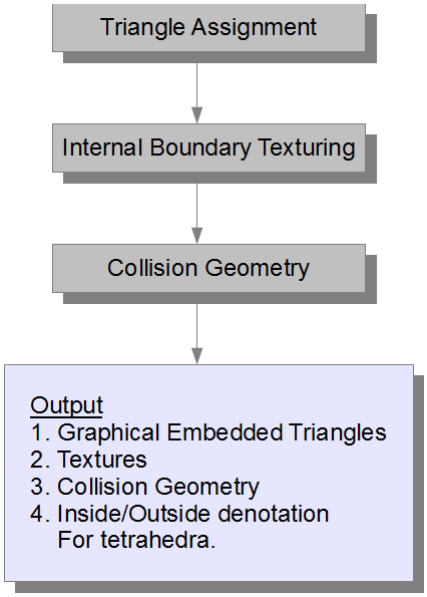


Fig. 3. High-level overview of preprocessing stages. Triangle assignment is first performed where each triangle is assigned to one ore more tetrahedra via its barycentric coordinates. Internal boundary texturing processes the tetrahedral boundaries which are inside the object and rasterizes them for future fracture boundaries. Collision geometry creates the geometry used in the physical collision detection/handeling.

Table I. Terminology associated with the preprocessing discussion.

T	The graphical triangle mesh.
T_k	A subsection of the triangle mesh belonging to physical tetrahedron k.
t	An individual triangle in the graphical mesh T.
P	The physical tetrahedral mesh.
p	A single tetrahedron of the physical mesh P.
f_i^k	The i^{th} triangular face of tetrahedron k where $i \in [1, ..., 4]$
β_{ij}	Barycentric coordinates of the j^{th} vertex of the i^{th} triangle in the graphical mesh T.

A set of common symbols will be used to denote which components of the mesh are currently being discussed. Table I lists the symbols and terminology associated with the following chapter.

The input to the preprocessing phase will be a triangle mesh T and a tetrahedral mesh P that encloses the triangle mesh. T must be a mesh free of holes and self-intersections. The mesh T simply consists of a collection of triangles defined by their vertices and faces. No adjacency information or connectivity information is required. The tetrahedral mesh P should be generated such that each primitive of T will be completely contained within one or more primitive of P. Not all triangles in T belong to exactly one tetrahedron p but instead might span a subset of P. To generate P, a coarse triangle mesh is constructed to completely enclose T. From this coarse triangle mesh, any tetrahedralization algorithm may be used to create internal connected tetrahedra. For this research, the package TetGen [33] was used in order to create P

In this chapter, the 3 main functions of the preprocessing phase will be explained. First, the triangle assignment procedure will be discussed. The assignment procedure

includes a discussion of how barycentric coordinates are used to assign triangles to a tetrahedron in the mesh. Following the triangle assignment phase, the texturing phase will be discussed. During the texturing phase, walls of each tetrahedron are partially textured based on whether they are inside/outside of the geometry. Finally, collision geometry generation will be discussed. The collision geometry phase is the most computationally intensive component of the algorithm. See Figure 4 for a high-level view of the preprocessing stage.

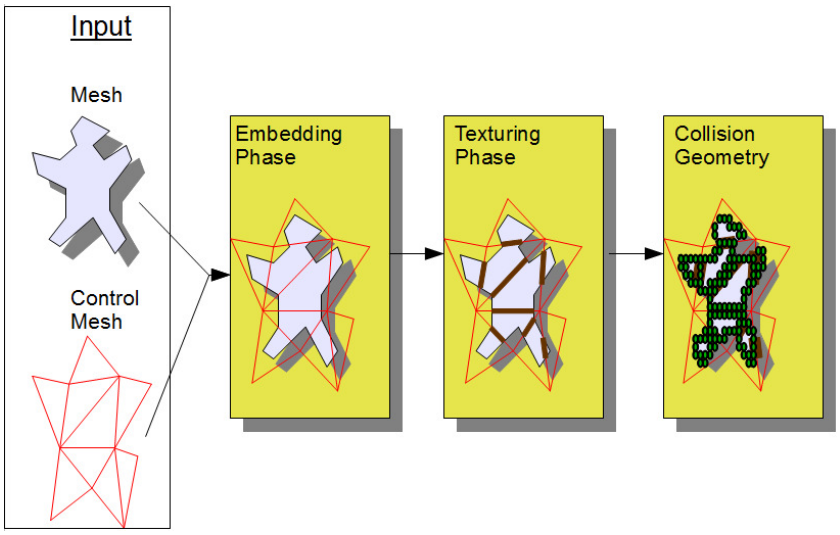


Fig. 4. Overview of the preprocessing steps. Input includes a polygonal mesh and a tetrahedral mesh. The polygonal mesh is embedded within the tetrahedral mesh. Texturing is then performed on the interior portions of the tetrahedral mesh. Finally, collision spheres are generated.

A. Barycentric Coordinates

Barycentric coordinates are used in this simulation to create a linear interpolation function and are used to perform the embedding procedure. Barycentric coordinates have the property that they are used to perform a convex combination of an individual

tetrahedron's vertices into one single point. The barycentric coordinates themselves can also be used to determine properties of the embedded points with respect to their embedder tetrahedron. Given the j^{th} vertex of triangle i defined as $t_{ij} \in \mathbb{R}^3$ with barycentric coordinates $\beta_{ij} \in \mathbb{R}^4$ with coordinates $[u,\ v,\ w,\ t]$, and the tetrahedral vertices p_0, p_1, p_2, and $p_3 \in \mathbb{R}^3$, the function used to determine the 3-D position of t_{ij} is

$$t_{ij} = \phi(u, v, w, t) = u * p_0 + v * p_1 + w * p_2 + t * p_3 \tag{3.1}$$

Barycentric coordinates have the property of affine invariance which, for the purposes of this research, intuitively means that as the tetrahedron deforms, each embedded point's new position will have the same barycentric coordinate as in the undeformed state. Affine invariance is important when considering deformation across boundaries of the tetrahedron. If a face of a single tetrahedron is connected (i.e. all the vertices of the face are co-incident with the vertices of another tetrahedron's face upon initialization), a single point that lies on the shared face that is actually contained in both tetrahedra will remain coincident as long as that face is still shared. During deformation (avoiding fracture) a surface crossing a shared face of the embedded geometry between two tetrahedra will remain C^0 continuous across this boundary. C^0 continuous means there will be no positional discontinuities in the geometry within the shared regions of the surface even though the normal across these regions may change discontinuously.

The calculation of barycentric coordinates is simple and straightforward. One way to compute the coordinates is to solve a linear system. The system can be set up as follows,

$$\begin{bmatrix} p_1 & p_2 & p_3 & p_4 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \beta_{ij} \end{bmatrix} = \begin{bmatrix} t_{ij} \\ 1 \end{bmatrix} \tag{3.2}$$

where p_i is the tetrahedron's i^{th} vertex defined as $[p_i^x, p_i^y, p_i^z]^t$, β_{ij} are the barycentric coordinates of vertex t_{ij} defined as $[\beta_{ij}^u, \beta_{ij}^v, \beta_{ij}^w, \beta_{ij}^t]^t$ with respect to the tetrahedron, and t_{ij} is the point's position defined as $[t_{ij}^x, t_{ij}^y, t_{ij}^z]^t$.

Both p_i and t_{ij} are given, so the system is solved for β_{ij}. Note the bottom row of the left matrix consists of 1s because the barycentric coordinates should all sum to 1. Assuming the tetrahedron is not degenerate (i.e. vertices are not coplanar or coincident), the system can be solved with any number of techniques. For this research, Cramer's rule was utilized to solve the system. For more information on the properties of barycentric coordinates, see Farin [34].

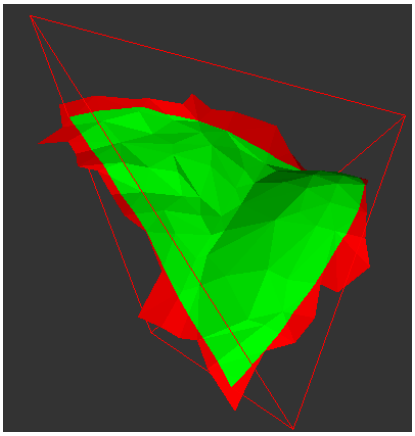


Fig. 5. Subsection of a mesh and a single tetrahedron. Red portions correspond to the triangles and sub-sections of triangles which are outside the tetrahedron, and green portions correspond to the sections which are inside.

Barycentric coordinates can also be used to determine whether a point is outside or inside the tetrahedron. Given t_{ij} and its barycentric coordinates β_{ij}, t_{ij} is inside

the tetrahedron if all 4 components of β_{ij} are positive. The point lies outside the tetrahedron if any of the components are negative. See figure 5 for an example of barycentric correspondence to a tetrahedron.

B. Triangle Assignment

Initially T consists of a collection of triangles defined by their vertices and their faces. No adjacency information for the triangles is required. This phase in the preprocessing procedure consists of iterating over all triangles t in T and assigning them to one or more p in P. In order to test for assignment, it is not sufficient to simply test whether a vertex belongs to a tetrahedron or not. A triangle may span multiple tetrahedra, or a portion of the interior of the triangle may be intersecting a tetrahedron. For each tetrahedron in the physical mesh P, each triangle t of T will be tested for possible inclusion.

A simple test between the triangle in question and all triangular faces f_i^k of the tetrahedron is first performed. The vertices of each t are first converted to barycentric coordinates with respect to the current tetrahedron. First, a simple intersection test is performed between the triangle t and the 4 triangular faces f_i^k of tetrahedron p. If t does not intersect p then an additional test is performed. The barycentric coordinates β_i^k of each vertex of t is computed and checked for inclusion inside p. As previously mentioned, a vertex of t is inside the tetrahedron if all of its barycentric coordinates are 0. If no vertices of t are found to be within p, then it fails the second test. A triangle that passes at least one test is embedded within p.

These two checks are performed for every triangle in P and every tetrahedron in T. While it could be argued that some form of spatial subdivision scheme should be used to cull out impossible intersections, no such scheme was used in this research

since performance is not an issue for the preprocessing step. Any triangle that is embedded within any tetrahedron gets added to that tetrahedron's list of included triangles. The barycentric coordinates of the triangle's 3 vertices are stored as well. These coordinates along with their tetrahedral counterpart are needed for display. It is important to note that a single triangle may get assigned to multiple tetrahedra and may be duplicated. Duplication only occurs on the boundaries of each tetrahedron. Obviously the number of duplicated triangles is determined by the number of tetrahedra, but typically the size of P will be very small compared to the size of T. Details on the numbers of tetrahedra and corresponding graphical triangles can be found in the results chapter. Duplicated triangles will not cause graphical artifacts since the portions of the triangles that fall outside of their tetrahedra are not displayed.

Once all of the triangles in T have been assigned to tetrahedra in P, the mesh is then checked again for empty tetrahedra. Empty tetrahedra occur due to alignment issues in the control mesh or from the generation of tetrahedra in the tetrahedralization algorithm. Empty tetrahedra represent portions of the control mesh which lie outside of the graphical mesh altogether.

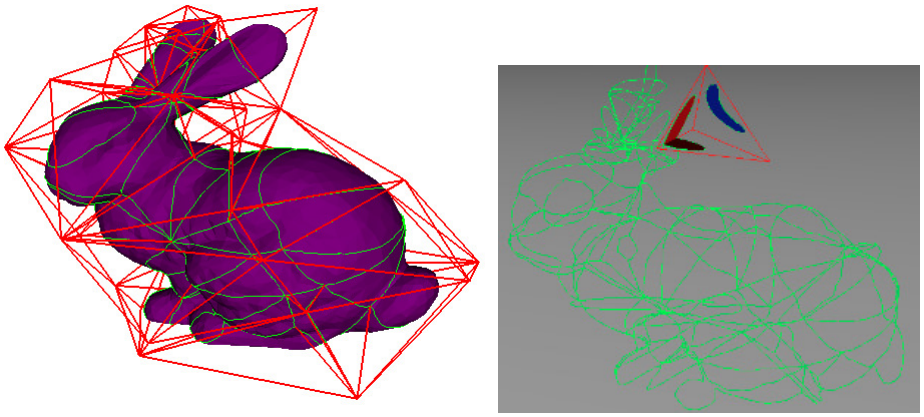
C. Texturing

A method to close off the boundaries of the tetrahedra was created. This method is not exact and contains small-scale gap artifacts when zooming close to the geometry, but in practice and real-time scenarios these artifacts are unnoticeable and can be alleviated with higher resolution textures.

The texturing phase uses the triangle lists generated in the previous step to paint the walls of the tetrahedra in P which fall within the physical solid object. The painting of the walls does not require any type of re-meshing and simplifies some GPU

based effects during graphical simulation. The textures that are created are the only graphical portions of P which the user will see during run-time. Each tetrahedron in P will reference exactly two square textures. Each texture will contain two faces of the tetrahedron.

A scan-line approach is taken to create the tetrahedral wall texturing. This approach takes advantage of the fact that T is a closed manifold mesh without any holes. For this reason, the intersections of the triangles and the tetrahedra will be completely connected across tetrahedral boundaries as can be seen in Figure 6.



(a) Bunny with interestions with tetrahedra.

(b) One tetrahedron being filled.

Fig. 6. The Stanford bunny model with an example tetrahedral control cage and the intersections between graphical and physical geometry. 6(a) The model and the intersection lines can be seen. 6(b) A single tetrahedron around the ear gets filled.

First each triangle of the embedded geometry will be intersected with its respective embedding tetrahedron. A set of contour lines are created from this intersection. Each contour line will belong to a face of a tetrahedron and is stored in a list.

Once all contour lines are computed from the intersections, all tetrahedral faces are rasterized using the scan-line algorithm. For the scan-line algorithm, each tetra-

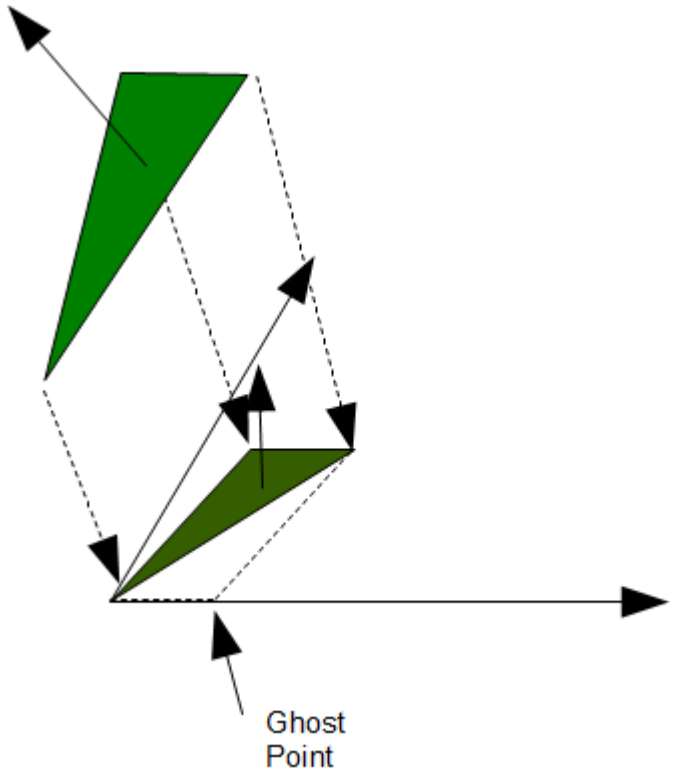


Fig. 7. The triangles are projected to 2-dimensional space in order to avoid numerical precision errors.

hedral face will be bi-linearly interpolated from its vertices. In order to perform bi-linear interpolation, a ghost point is computed for the tetrahedral face which will cast the face into a triangular half of a quad. Given f_i^k representing the i^{th} face of tetrahedron k, the vertices of the face can be represented as v_i with $i = 1, ..., 3$ in a counter-clockwise ordering. Let $v^1(u, v) = s$ where $s \in \mathbb{R}^3$ lies on f_i^k if $0 \leq u \leq 1$ and $0 \leq v \leq 1$. Also, define a ghost point $v_g = v_1 + (v_2 - v_3)$ (see Figure 7). Then

$$v^1(u, v) = (1 - v) * ((1 - u) * v_1 + u * v_g) + v * ((1 - u) * v_3 + u * v_2) \tag{3.3}$$

is a bilinear interpolation for the first half of the texture where $v^1(0, 0) = v_1$, $v(1, 1) =$

v_2, and $v(0,1) = v_3$. For the second half of the texture, the same function is defined with u and v reversed,

$$v^2(u,v) = (1-u) * ((1-v) * v_1 + v * v_g) + u * ((1-v) * v_3 + v * v_2) \tag{3.4}$$

and will give the values of $v^1(0,0) = v_1$, $v(1,1) = v_2$, and $v(1,0) = v_3$. By casting the triangle problem into that of a quadrangular interpolation problem, the faces can be evenly interpolated in a single loop over u and v.

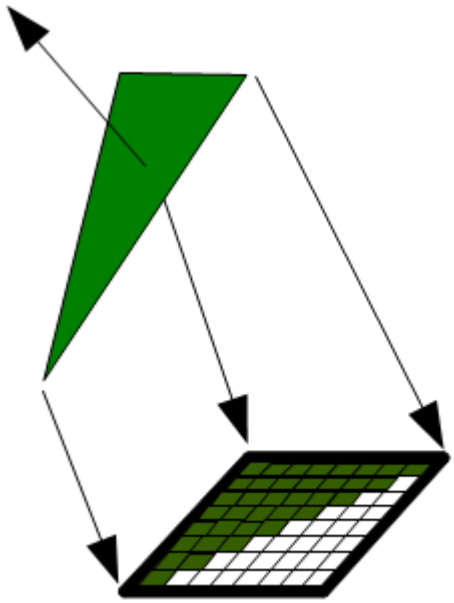


Fig. 8. A triangle is stored as one half of a square texture.

Starting at one vertex of the triangle, the algorithm loops through the u and v coordinates. This loop means to increment u and v by du and dv respectively. du and dv are dependent on the width and height of the target texture and are computed as $du = \frac{1}{w}$ and $dv = \frac{1}{h}$ where w and h are the width and height of the texture, respectively. Figure 8 shows an example of a triangle being rasterized in a 2D-texture space. Upon each increment of u or v, the contour lines from the current

triangle are checked for intersection with the line segment formed from the current and previous u and v. An approach common to computer graphics is adapted for use in this determination. Consider a 3D polyhedron which is closed and manifold. Given a single point, a ray can be cast in any direction. If the ray crosses an odd number of faces of the polyhedron then the point is inside the space enclosed by the polyhedron. If the ray crosses an even number of faces of the polyhedron then the point is outside the space. This approach to using a ray cast to perform an inside/outside test on a polyhedron is used in 2-dimensions here: instead of a polyhedron we have a set of line segments in 3-dimensional space and the rays are scan-lines marching over the face of a tetrahedron (within the same plane as the face). The situation is made a bit more difficult since an explicit edge is not computed for the tetrahedral facet's boundaries. The edges of the tetrahedral face can not simply be used either as it creates a possible arbitrary assignment for the point (see Figure 9). More information about the edges must be known to create a true boundary for the contour lines.

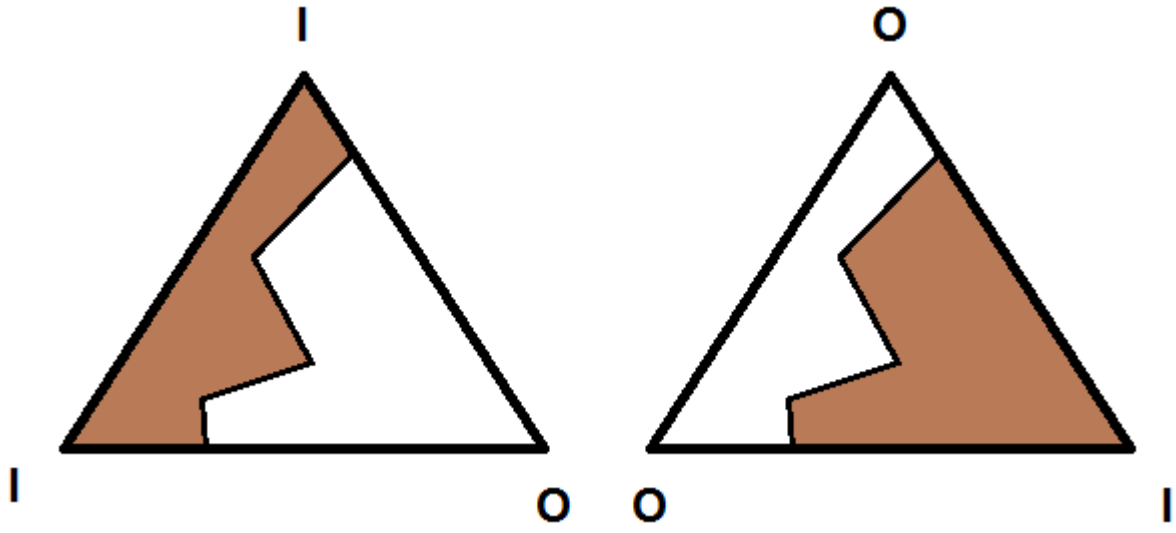


Fig. 9. Two possible configurations for inside/outside along a face. Using information from the vertices is required.

Instead of closing off the contour lines, a different approach is taken. In this approach, an initial inside/outside denotation is performed for the vertices of each tetrahedron. These vertices can be considered inside or outside by using the method discussed previously along with the 3D mesh T. With the inside/outside information for the vertices of the tetrahedrons, it is possible to begin at one vertex and scan across an individual tetrahedron's face. Initially the current state is set to either inside or outside depending on the starting vertex. While marching over each tetrahedron's face (using the interpolation scheme previously discussed), each intersection of the marching ray with the contour lines will swap the current state to "inside" or "outside". A counter is used and the odd/even scheme is used in the case that du or dv is sufficiently large enough to cross multiple isolines. This process is performed for every texel in every tetrahedral face to create the wall textures. For an example, see Figure 10.

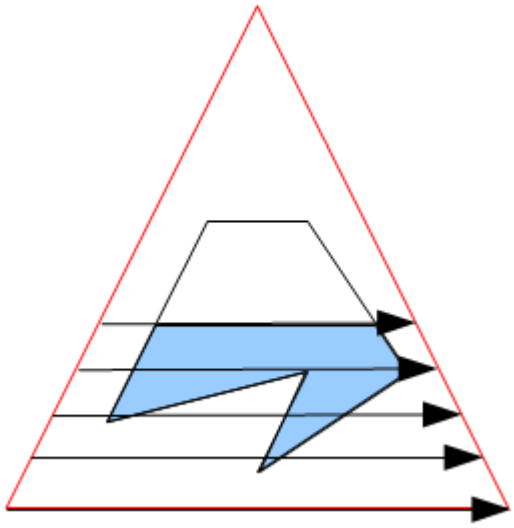


Fig. 10. The scan-line approach fills in texels one-by-one by marching from a vertex assigned as inside/outside and keeping track contour intersection.

The marching algorithm attempts to compare 3-dimensional line segments with

other 3-dimensional line segments. In order to avoid numerical precision errors and epsilon-like computations, the tetrahedral faces are first projected to 2D. For each tetrahedral face, there is a choice for choosing the projection dimensions from a 3- dimensional space. The 2 dimensions chosen are the dimensions which have the smallest absolute value components in the face's normal. This projection is motivated by image collision techniques found in Faure et al. [35]. Once the faces of the triangles are projected to 2D, the problem of intersecting the 2D line segments becomes easier.

The color stored for the current texel is given an alpha component of 1 if it is inside the contour region and 0 if it is outside. Once all of the faces of the tetrahedra are rasterized, the textures are then stored in memory. Before the simulation begins, the textures are used with alpha blending enabled and the alpha culling function set to "greater than 0". Because the texels which were determined to be outside the contour regions have alpha components of 0, these will be culled out by the graphics API pipeline. The portions which are set as inside have an alpha component of 1 and will be displayed during animation. The other colors in the texture are arbitrary and may be used whatever the developer desires, such as a normal for a normal map.

D. Collision

The model so far consists of a set of triangles for graphical display, a set of tetrahedrons for physical simulation, a set of textures which will be painted onto the faces of the tetrahedrons for graphical display, and a set of mappings from triangles to tetrahedrons. In order to have a simulation which contains any interesting effects, a method to perform collisions must be created.

1. Image Based Collisions

Initially a well-known image-based collision technique was researched and implemented for this problem. The technique is attributed to Faure et. al. [35]. The image-based collision approach would correspond well to this technique since there are portions of the graphical mesh which are implicit, i.e. the painted textures. In the image-based approach, an object is drawn from three orthogonal directions. A layered-depth image (Shade et al. [36] and Everitt [37]) is used to perform multiple renders on different layers of the geometry in screen space. From these layered depth images rendered from 3 orthogonal directions, intervals can be created which represent the interior of the object. Using these intervals, collision detection can be determined as well as the associated collision response. The collision response includes using the relative positions of the interval endpoints and mapping the gradient of the penetration volume to the object's exterior vertices. While this approach works well for solid deformable objects, it was found to suffer from multiple deficiencies with the inclusion of fracture. When pieces of the object break away, the bounding box used to render the object could grow very large. Due to the discretization inherent in rendering to a texture/framebuffer, precision is lost as the bounding box grows larger. Furthermore, performing multiple passes on separated pieces of geometry is a possibility, but each piece of the object would have to be rendered multiple times. Obviously this is inefficient and undesirable. While other image-based collision approaches were considered, none were found sufficient to deal with geometry breaking apart into multiple geometry. The development of an image-based collision scheme with a fracture simulation is an area of possible future research.

2. Collision Spheres

Instead of using an image-based approach, a particle-centric approach was decided upon. Using a particle-centric collision method allowed the collision detection to remain simple and efficient. While the particle-based approach may allow interpenetration of the graphical mesh, the visual results are negligible in a real-time scenario.

A bounding box is first computed for every tetrahedral element in the physical mesh P. An initial target radius is chosen by the user. This radius affects the maximum bounds on the particle radii which are computed. Depending on the radius chosen, the volume of the box is looped over in discrete intervals. At each interval, the current position is tested for containment inside the current element. If the current position is found to be inside, a particle is added to the current tetrahedron and barycentric coordinates are assigned to the particle with respect to that tetrahedron. If, during the scan of a tetrahedron, at least 3 particles are not created, then the radius is halved and the tetrahedron is rescanned. The reason for decreasing the radius length and rescanning the volume is due to the error inherent with a discretized volume. The 3-particle minimum criteria was selected since 3 points are required to create a plane. With less than 3 points, pieces of the object may never settle when laying stationary on the ground. After particles have been assigned to their respective tetrahedra, they are rescanned for possible exclusion. If a particle is found to be completely surrounded by a certain amount of neighbors, it is deleted from the collision volume. For this research, 26 was used as the culling number. The spheres are culled since the simulation avoids any re-meshing of the individual tetrahedra and thus collision particles which are completely surrounded by other particles will never collide with external geometry.

From the initial particle assignment, a center of mass for each tetrahedron can

computed. In this research, the mass for a tetrahedron corresponds to the fraction of the total volume the particle spheres account for and the total volume of the tetrahedron. The center of mass itself is also be stored barycentrically for each tetrahedron and the mass computed is distributed to the tetrahedron's nodes via these barycentric coordinates.

CHAPTER IV

PHYSICAL FORMULATION

In this chapter, the fundamental equations relating elasticity and strain will be shown as well as how deformation forces may be derived from these properties. For this discussion, general fracture will be ignored and instead the deformation of the object will be considered. The equations relating to deformation were derived by the early work of Terzopoulos et al. [4] and subsequently extended to handle fracture by O'Brien and Hodgins [2]. From a high level point of view, the simulation begins with a tetrahedron mesh which will represent the domain on which the equations of motion will be derived. The simulation is then treated as an initial value problem. A time-step is chosen which is used to update the object's position, velocity and internal/external forces. Without regarding any external environment, the object will simply continue to fall due to gravity and will eventually hit terminal velocity if air forces are used. If the object comes into contact with the environment, an inter-penetration constraint moves nodes which have crossed physical boundaries back to the surface of the boundary. Using the FEM, equations will be integrated over the tetrahedra which will give rise to internal forces acting on those nodes. Intuitively, these forces will be greater the larger the deviation from relative rest state. During each time-step, internal forces within the object will be integrated and subsequent velocities and positions of the nodes will be modified based on Newton's second law of motion.

A. Overview of Finite Element Method as Applied

Only recently have truly massive destructible interactive environments been implemented in production quality software. Many of the destruction models make use of

a simplified finite element method (FEM) simulation. The finite element method has been employed extensively in mechanical engineering and numerical analysis due to its properties of being very amenable to algorithmic computations. The main idea behind the FEM is to use a finite approximation of a surface or volume in order to solve some function over the domain of that object. The simplest way to define the function over these elements is linear interpolation. Multiple basis functions can be chosen for each element such that there is a basis function which computes to 1 at an element node and falls to 0 for all other nodes, and there is one such basis function for each node in the finite element mesh. Other functions can be defined in terms of these basis functions which leads to a piecewise linear approximation of the function being evaluated.

By defining a basis matrix for each element, any function can be interpolated over the volume of the tetrahedron. Stresses and strains were evaluated over the finite element volume which were derived from the relative changes from rest shape to current simulation shape. Deformations arising from the deviation from rest state create stresses and strains within the material of the object. From these forces a fracture plane is computed which represents a failure in the material. The failure in the material gives rise to a fracture, a.k.a a discontinuity in the material. The evaluation of these forces/fractures could then be repeated at discrete time-steps as is typical of physically-based simulations. The model presented by O'Brien and Hodgins [2] and Parker and O'Brien [3] will be used for this work.

B. Elastic Deformation

Elastic deformation occurs in an object when a deviation from the relative rest state to the current positions of an object changes. Purely rigid body displacements (trans-

lations and/or rotations) do not change the relative configuration of an object's nodes with the other nodes and thus do not result in a deformation of the object. In the case that the relative positions of an object do change, internal forces will arise. These internal forces are equalization forces which attempt to "undeform" the object back to its rest state.

The rest positions of the object can be represented by u_i where $u \in \mathbb{R}^3$ and i is the i^{th} position of the object. In this work, the rest position is determined by the initial position of the object. The rest position will not change throughout the entire simulation. The current positions of the object are represented by x_i where $x \in \mathbb{R}^3$ is defined as u. It is from these two components of the object that the stress/strain relationships are derived.

C. Deformation Gradient

In order to compute internal forces acting on the object, a deformation gradient is computed for each element. Since the finite element method is a piecewise linear method, using only one deformation gradient per element corresponds to the linear nature of the domain.

A deformation gradient is a 3 x 3 matrix describing how the element is deforming with respect to its material rest coordinates. The deformation gradient itself is computed regardless of whether the transformation is rigid or deformable. In order to compute the deformation gradient, a basis matrix must be computed for the tetrahedron. The basis matrix is a 3 x 3 matrix and is computed using the material coordinates of the tetrahedron. Each column of the basis matrix is a vector representing an edge in material space of the tetrahedron. All column vectors will originate at the same vertex. A graphical visualization can be seen in Figure 11. More formally

a matrix Du will be defined as

$$Du_j = u_j - u_0$$

where j is a 3-component column of the matrix Du.

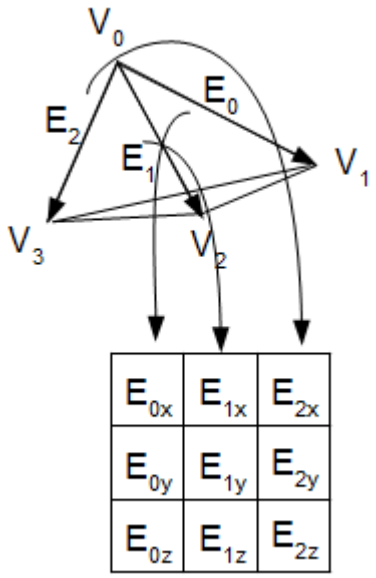


Fig. 11. An example of how the basis matrix is built from the relative configuration of the nodes in a tetrahedron.

Using Du the basis matrix β is defined as Du^{-1}. The deformation gradient is computed from a linear combination between the basis matrix and a matrix Dx which can be computed similarly to Du. The deformation gradient can be defined as

$$F = \frac{\partial x}{\partial u} = D_x \beta \tag{4.1}$$

The co-rotational formulation is used to extract a rotation from the deformation tensor since Cauchy's infinitesimal strain tensor is not invariant to rotations. Following Etzmuß et al. [18], Muller and Gross [19], and Parker and O'Brien[3] among others, polar decomposition is performed on the deformation gradient F. Polar decomposition will factor $F = QA$ where Q is an orthogonal matrix and A is a positive

semi-definite symmetric matrix. Q represents a rotation of the element from rest shape, and A represents the amount of stretching/shearing of the element. Since Cauchy's tensor is a linearized strain tensor, factoring out the rotation such that

$$\hat{F} = Q^T F \tag{4.2}$$

will provide the deformation gradient in unrotated space (see Figure 12 for an example of the rotation being factored out). Further calculations will require the factored deformation gradient $\hat{F}$ and the 3 x 3 matrix Q. Since Q is orthonormal, equations involving Q^{-1} can instead be computed simply using Q^T which will allow for more efficient computations.

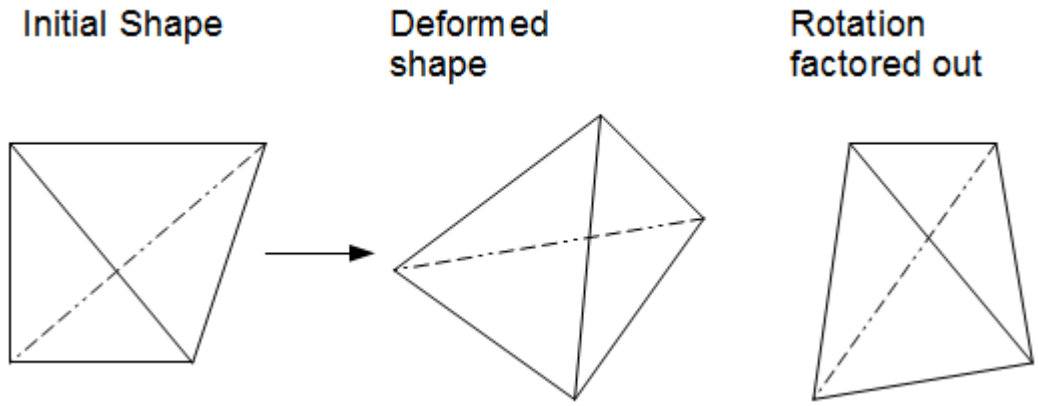


Fig. 12. Rotations are factored out of the deformation gradient in order to represent pure relative displacements from rest shape.

D. Strain Tensor

From the previous formulation, a strain tensor and stress tensor can be computed. The strain tensor describes the linear relationship between how a portion of the object is being deformed with respect to its rest shape. The strain tensor used in this research is Cauchy's infinitesimal strain tensor which is not invariant with respect to rotation.

As will be shown in a subsequent section, a rotation of the deformation gradient will allow us to use a linear strain tensor instead of a higher order tensor. Cauchy's tensor is defined as

$$\varepsilon = \frac{1}{2}(\hat{F}^T + \hat{F}) - I \tag{4.3}$$

where $\hat{F}$ is the deformation gradient with rotations factored out which can be computed as in the previous section and I is the identity matrix.

E. Stress Tensor

The stress tensor can be computed as

$$\sigma = \lambda Tr(\varepsilon)I + 2\mu\varepsilon \tag{4.4}$$

where λ is Lame's first parameter and μ is the shear modulus. Equation 4.4 is a 3D generalization of Hooke's law for linear isotropic materials. The stress tensor of the material can be used to compute the forces that a single tetrahedral element exerts on one of its nodes. The force is defined as

$$f_i = Q\sigma n_i \tag{4.5}$$

where f_i is the force exerted on node i of the tetrahedral element and n_i is the area-weighted outward normal of the face opposite of node i.

F. Time Integration

Originally, explicit time integration schemes were tested using the forces computed directly from equation 4.5. Both simple Eulerian time integration and Runge-Kutta

4 were tested. While Runge-Kutta 4 provided a substantial increase in stability over the explicit Euler method, both required restrictive requirements on the time step dt which limited real-time performance.

The choice was made to use an implicit integration scheme following recent work by Parker and O'Brien [3] since implicit integration schemes provide unconditional robustness using any size time step. While any size time step could be used, smaller time steps typically provide much more vibrant simulations. For this research, the time step of the simulation uses a variable time step method. Implicit time stepping methods allow for arbitrarily large time steps at the expense of solving a linear system. The derivation used in Baraff and Witkin [6] and subsequently Parker and O'Brien [3] was used in this research. For a full treatment and explaination of the implicit time stepping method, see Baraff and Witkin's research publication [6].

For implicit integration, a global stiffness matrix must be computed for the entire system of nodes. The stiffness matrix of each individual element is first computed and these are assembled into the global matrix.

For a given element, the stiffness matrix can be computed from the set of Jacobians within the element. A set of 4 3x3 Jacobian matrices are computed for each node in the tetrahedron (a total of 16 matrices per tetrahedron). The Jacobian is the set of partial derivatives of the force exerted on one node with respect to the position of another node.

$$J_{ij} = -Q(\lambda n_i n_j^T + \mu(n_i \cdot n_j)I + \mu(n_j n_i^T))Q^T \tag{4.6}$$

where J_{ij} is a 3x3 matrix of the Jacobian of a force on node i with respect to the position of node j, n_i is the normal of the face opposite of node i, λ and μ are Lame's constants relating to material properties, and Q is the matrix factored out of

the deformation gradient by the polar decomposition step.

The stiffness matrix of an element is a 12x12 matrix which contains all 16 3x3 Jacobian matrices of element. Specifically,

$$K_i = \begin{bmatrix} J_{00} & J_{01} & J_{02} & J_{03} \\ J_{10} & J_{11} & J_{12} & J_{13} \\ J_{20} & J_{21} & J_{22} & J_{23} \\ J_{30} & J_{31} & J_{32} & J_{33} \end{bmatrix} \tag{4.7}$$

where K_i is the stiffness matrix for the ith element.

A global stiffness matrix, K, can be computed from the Jacobians of a node. Assembling a stiffness matrix from individual stiffness matrices is standard practice in the finite element method. Individual entries in the stiffness matrix will be summed up if multiple tetrahedra share an edge with each other. The global stiffness matrix K is defined as

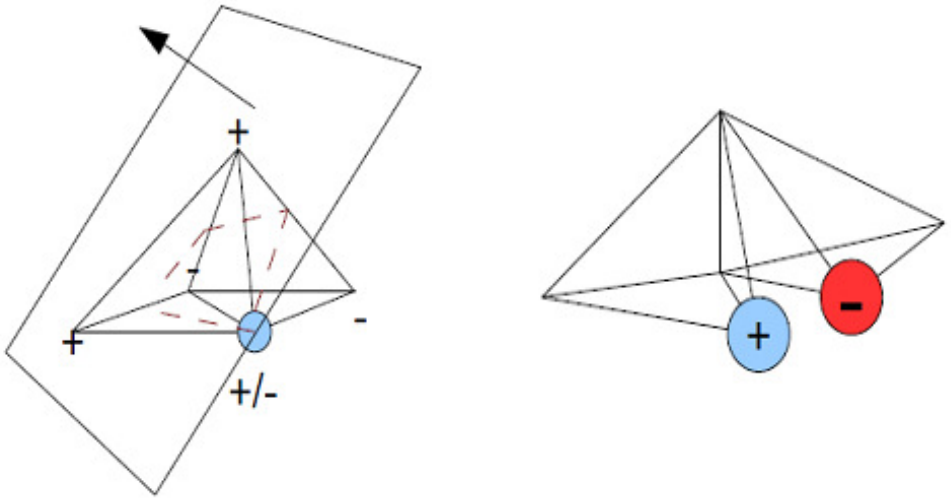
$$K = \begin{bmatrix} J_{0,0}^{tot} & J_{0,1}^{tot} & \cdots & J_{0,n-1}^{tot} \\ J_{1,0}^{tot} & J_{11}^{tot} & \cdots & J_{1,n-1}^{tot} \\ \vdots & \vdots & \ddots & \vdots \\ J_{n-1,0}^{tot} & \cdots & \cdots & J_{n-1,n-1}^{tot} \end{bmatrix} \tag{4.8}$$

where $J_{i,j}^{tot}$ is the sum of all the Jacobians of the ith node with respect to the jth node. The sum comes into play when two tetrahedra share an edge.

G. Fracture

The computations of the fracture planes are very straight forward and can be derived from the strain and stress tensors. Following O'Brien and Hodgins [2], a separation tensor is computed for each node from the tensile and compressive forces acting on

that node. The eigenvalues and eigenvectors of the separation tensor will determine if there is to be a fracture at the node and how the fracture plane is oriented. For this simulation, the fracture plane is not used to explicitly drive crack propagation. The plane itself is used to split a single node into two nodes. The tetrahedra that are connected to the split node are assigned to one of the new nodes based on their orientation with respect to the fracture plane. Fracture will cause discontinuities on tetrahedral boundaries only (see Figure 13).



(a) A fracture plane is computed. (b) Nodes split.

Fig. 13. An example of a fracture occurring between two tetrahedra. The plane is computed, and then forced to conform to the boundary of the tetrahedra. The fractured node is split into positive and negative nodes. Nodes on the positive side are assigned the positive split node, and negative nodes are assigned the other split node.

1. Computing Tensile and Compressive Forces

The tensile and compressive stresses are due to expansion of an element and the contraction of the element, respectively. The fracture method is derived from approximations stemming from fracture mechanics. For more information about the theoretical mechanics involved, see Anderson [25]. For information about the approximation used in this research, see O'Brien et al. [2].

2. Separation Tensor

Once the tensile and compressive stresses are computed they can be used in order to create a separation tensor at each node. The separation tensor can be used to find the plane of separation.

Let the separation tensor be ζ. The tensor represents the balanced tensile and compressive loads at a node in the object. From the tensor, a determination for whether there is material failure can be made. Before beginning, a function must be defined which is a outer product of a vector followed by a scaling of the vector's norm.

$$m(v) = \begin{cases} vv^T / \parallel v \parallel_2 & if \parallel v \parallel_2 \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.9}$$

Where v is an arbitrary vector. This matrix has the properties that its eigenvector is v normalized, and its eigenvalue is $\parallel v \parallel_2$. Using this matrix formulation and the tensile/compressive force set on a given node, the separation tensor can be computed as

$$\zeta = \frac{1}{2}\left(-m(f_{tot}^{+}) + \sum_{i=1}^{n^+} f_i^{+} + -m(f_{tot}^{-}) - \sum_{i=1}^{n^-} f_i^{-} +\right) \tag{4.10}$$

f_{tot}^{+} and f_{tot}^{-} is the sum of all tensile forces can compressive forces respectively at

the node, and n is the number of total tensile/compressive contributions at the node.

Eigen decomposition is performed on the separation tensor. Separation tensors which have an eigenvalue greater than the material strength of the node will fail at that node. The separation plane will be perpendicular to the largest eigenvector. When the material fails, the node will be split into two separate nodes which will be on each side of the separation plane. The above method deviates from O'Brien and Hodgins [2] since it does not re-mesh a tetrahedron by cutting it with the separation plane. Following the idea presented in Parker and O'Brien [3], the simulation is greatly accelerated by not re-meshing individual elements of the mesh, but instead constraining fracture and crack propagation to the boundaries of individual elements. While re-memshing is appropriate for simulations where a solution as close to exact as possible is desired, this research is concerned only with an approximation appropriate for interactive scenarios. Once the nodes are separated, the simulation continues with the newly added nodes. It is worthwhile to note that no nodes are ever removed throughout the entire simulation.

H. Collision Handeling

Collisions with geometry are handled during the simulation using the collision geometry computed during the preprocessing stage. Initially, collisions were confined to be on the faces of the tetrahedra. Obviously this constraint produces noticeable artifacts when empty portions of the geometry appear to be solid. The collision spheres computed in the preprocessing phase allow collision detection to only be derived from a more accurate representation of the embedded geometry without using the embedded geometry explicitly. Using spheres as the collision geometry greatly simplifies the equations relating to collision detection and handling. This simplified

model allows certain interpenetrations of the graphical models to occur, but the interpenetrations are small. Furthermore, real-time environments involving scenarios of high-impact physics often use approximation models similar to this in order to not bog down the speed of the simulation. Fast collision handling and detection for high-resolution models is an active area of research. As mentioned in the preprocessing section, image-based collision methods were first used to help speed up collision detection since portions of the geometry (painted walls of the tetrahedral mesh) are not part of the physical model and thus can not be used in the collision detection algorithm without first being rasterized in some way. Image-based collision techniques break down when multiple geometries are allowed to become arbitrarily far apart. The accuracy of the collision is inherently based on the resolution of the screen-space rasterization process. While loss of accuracy may be acceptable for graphical effects, collision detection affects the physics of the simulation and a loss in accuracy could lead to unacceptable scenarios for the user.

The collision system that was used in the research is based on impulse-based forces. When a sphere collides with an object an instantaneous change in its owning tetrahedron's velocity and position will occur. Whether a sphere collides with another sphere, a plane, or any other object a vector is computed which corresponds to the normal of the collision surface which is being collided with.

Given a sphere S with barycentric coordinates S_b with respect to tetrahedron T with vertex velocities V_i where $i \in [x, y, z]$, the velocity V^s of the embedded sphere can be computed as a barycentric combination of the tetrahedral node velocities. Given a normal N which is the normal of the surface the sphere comes into contact with, the normal component and tangental components of the collision can be determined as

$$S_n = (N \cdot S_{pos})S_{pos} \tag{4.11}$$

$$S_t = S_{pos} - S_n \tag{4.12}$$

with S_n and S_t being the normal and tangental components, respectively, and S_{pos} being the sphere's real 3D position computed using barycentric coordinates. An offset vector is computed which will move the sphere back to the surface of the colliding object (disregarding the tetrahedron it is embedded within) and the formula for doing so is

$$s^o = N * -S_n \tag{4.13}$$

where s_t^o is an offset vector for node t of the tetrahedron and can be distributed to the vertices of the embedded tetrahedron. There may be multiple sphere collisions in one time step, so for each t in the tetrahedron, the s_t^o giving the maximum euclidean norm $\|s_t^o\|$ arising from all the spheres will be used. Once all spheres in the tetrahedron have been tested for collision, the final offset vector is computed as

$$t^o = max(s_t^o) \tag{4.14}$$

where $max(s_t^o)$ is the maximum offset vector on node t. The index of the sphere corresponding to the contribution for each node is stored for use in the elastic/friction calculations. Offsetting the tetrahedral nodes will result in an instantaneous change in the tetrahedral nodes' positions. Since the finite element model is being used, this change will give way to stress and strain forces within the object. The velocities of the nodes are also offset due to elastic/friction reactions. These reactions are calculated as is typical of physically based particle simulations (see Witkin and Baraff's 2001 SIGGRAPH course notes [38]). The computation of forces arrising from collision with the surface make use of the velocity of each sphere.

CHAPTER V

GRAPHICAL SIMULATION

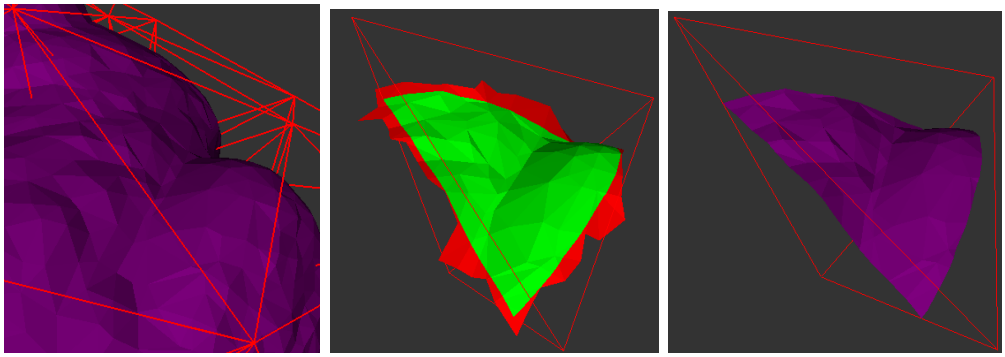
The graphical portion of the simulation renders the graphical model in conjunction with the graphical portions of the physical model in a way that makes the user believe the object itself is deforming and breaking apart. The actual graphical geometry is first offloaded to the GPU after the preprocessing stage. Modern graphics cards support multiple types of memory for use with storing display data. While the types of memories vary from card to card, the two most common types of memory types are static and dynamic. Static portions of the graphics card memory are used for geometry which will not be changed through API calls during the simulation. Dynamic areas are portions which may be modified frequently during the coarse of the simulation. Since the the embedded triangles do not need to be re-meshed during the simulation, they can be stored in the static area of memory on the graphics card.

Embedded triangles are stored as a vertex buffer object consisting of 4-component vertices. The 4 components of each vertex are the barycentric coordinates of the vertex with respect to its owning tetrahedron. During the display of the triangle, the vertex shader will use the barycentric coordinates to transform the owning tetrahedron's vertices into a single vertex of the embedded triangle. Physical tetrahedra are stored as vertices $\in \mathbb{R}^3$ in a dynamic memory location of the graphics card. During the simulation, API calls will be used to update the graphical tetrahedra with the data from the current frame of the simulation. The vertices of the tetrahedra array are accessed during embedded triangle position computation. Because each embedded triangle must know which tetrahedron to use, an attribute for each triangle consisting of the owning tetrahedron's index is passed to the graphics pipeline as well.

A. Discontinuities

Simply drawing all embedded triangles is not enough, even when fracture is not involved. Triangles along the boundaries of the tetrahedra are duplicated for each tetrahedron they are contained within since each instance of the triangle in each tetrahedron requires a different set of barycentric coordinates. Artifacts could occur during deformation which will cause the triangles to appear to be discontinuous and jagged along the the boundaries. Furthermore, triangles would appear to penetrate each other during fracture which is obviously not ideal. Instead of displaying the triangles as-is, the triangles are clipped to the boundaries of the tetrahedron they are embedded within. Modern graphics APIs allow user-defined clipping planes which will restrict regions of the 3D space from being displayed. This approach was tested first while drawing each set of triangles by first setting the clipping planes to be the boundaries of the tetrahedral elements before drawing the triangles contained within those sets. Setting the clipping planes of the simulation amounts to a state-change in the underlying graphics API which is inefficient to perform for every element in the simulation.

A simpler and more efficient alternative to using clipping planes is to exploit the interpolation abilities of the graphics pipeline in conjunction with alpha culling in order to discard fragments of the graphical triangles which would be clipped by their owning tetrahedron. Attributes assigned to vertices of a triangle are automatically interpolated on a per-fragment basis for use in a fragment shader. Color interpolation is the simplest example of this technique. Color values are assigned to each vertex of the triangle, and along the interior of the triangle the colors are linearly interpolated between the vertices. For more information on the rasterization properties of modern graphical hardware, see Foley [39]. Barycentric coordinates themselves represent



(a) Section of a Mesh (b) Barycentric Coloring (c) Final Display

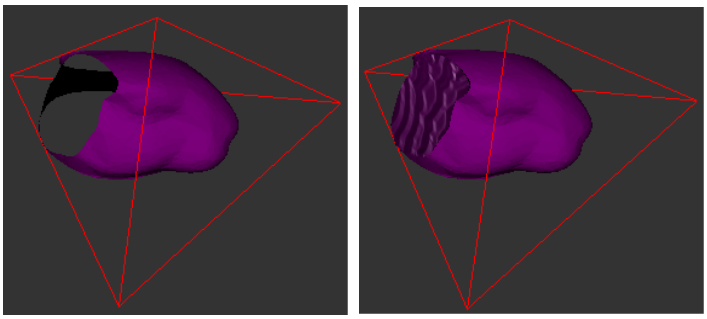
Fig. 14. An example of how barycentric coordinates are used. 14(a) A section of the mesh with control tetrahedra. 14(b) Triangles which intersect tetrahedron. Red portions correpond to the triangles and sub-sections of triangles which are outside the tetrahedron. 2(c) The final display triangles. Only sections inside the tetrahedron are displayed.

attributes at vertices of the triangles and will be interpolated over the face of the triangle. The barycentric coordinates for each vertex of a graphical triangle is passed through from the vertex shader to the fragment shader. During the execution of the fragment shader, the barycentric coordinates of the current fragment are checked for inclusion within the owning tetrahedron. The check consists of a simple conditional statement on whether each barycentric component falls within the range [0, 1]. If any component falls outside this range then the alpha value of the fragment is set to 0. If all components pass the unit interval inclusion test, the alpha value for the fragment is set to 1. Before displaying the graphical mesh, alpha culling is turned on with a function of "greater than 0" set. Alpha culling will not allow fragments to be written to the frame buffer if they fail the alpha culling function. Thus, using linear interpolation in conjunction with the barycentric coordinates of each triangle's

vertices and alpha culling enables the triangles to be properly clipped to their owning tetrahedron's boundaries. See figure 14 to see how triangles belonging to one tetrahedron are displayed.

B. Texturing

Since portions of the tetrahedral faces may be displayed along with the display mesh, the triangles of the tetrahedral mesh are used with the textures that were computed during the preprocessing phase. Due to the approximative quality of the fracture plane computation, the resulting elements all appear to be very linear and flat along the boundaries. The approximation presents artifacts where the user can see the tetrahedral boundaries, as well as whole tetrahedra for portions of the physical mesh which are completely inside. In order to alleviate these artifacts, normal mapping and displacement mapping are used to make the mesh appear to have a non-flat surface. The surface properties are derived from a normal map or a displacement map and can be used to both shade the surface and displace graphical geometry on the surface. Normal mapping is a well-founded technique and for a general survey of displacement mapping and normal mapping techniques the reader is encourage to see Szirmay-Kalos and Umenhoffer [40]. In order to perform normal mapping, a set of textures must be first generated with the desired material properties before the simulation begins. The displacement textures and/or normal maps are included with the assets to be loaded for the simulation. The textures could be either a 3D texture which would be accessed from the shaders, or it could be one or more textures which are assigned to different tetrahedron faces. See Figure 15 for an example of a disconnected mesh without and with texturing.



(a) No Texture (b) Texture

Fig. 15. An example triangle embedded inside two tetrahedral elements. 15(a) A portion of the mesh without texturing. 15(b) A portion of the mesh with texturing.

1. Normal Mapping

Boundaries of the mesh which are being intersected by geometry are displayed using normal mapping. Normal mapping is a well-founded technique which manipulates the normals along the surface in order to imitate detailed geometry on the face of a primitive. In this work, normal mapping is used on the faces of tetrahedra which are being intersected by the graphical mesh in order to give the face a more detailed look. See Figure 16 to see the difference.

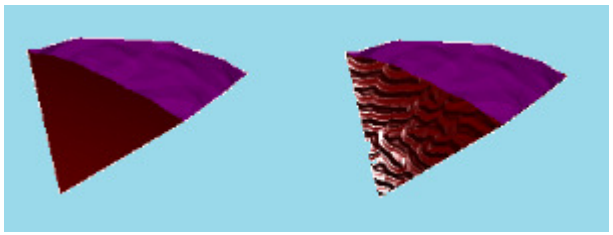


Fig. 16. Left: The interior portion of the surface without normal mapping. Right: The surface is rendered with normal mapping. The effects of normal mapping are enhanced via specular shading as can be seen in the image.

C. GPU Tessellation

Recent advances in GPU technology have allowed for dynamic tessellation of geometry on the GPU. Three new shader stages are included in the OpenGL graphics pipeline in addition to the common vertex and fragment shaders: the geometry shader, the tessellation control shader, and the tessellation evaluation shader. See Figure 17 to see the flow of the graphics pipeline with respect to the shader stages.

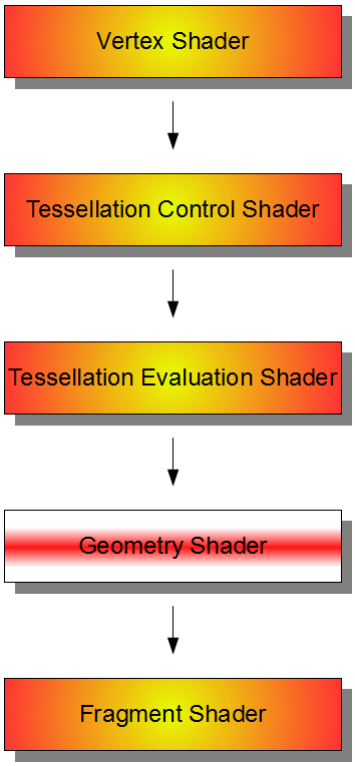


Fig. 17. The order of shader execution within the OpenGL graphics pipeline. Note that only programmable shaders are shown. The geometry shader is not used in this research.

The tessellation control shader and the tessellation evaluation shader were utilized to provide displacement mapping on tetrahedral boundaries which are completely contained inside the graphical object (no intersections of the graphical mesh along the surface of the face). Both of these shaders are designed to operate on

an OpenGL patch primitive. A patch is simply a collection of vertices defined by their 3-dimensional positions and are designated as either triangular or quadrangular patches.

1. Tessellation Control Shader

The tessellation control shader takes as input the patch defined by its vertices and performs some type of transformation on these vertices, similar to the vertex shader. The main difference between the tessellation control shader and the vertex shader is that it must set level-of-detail parameters for triangulation, and it is able to see all of the other vertices that are part of the patch. The purpose of the control shader is to first transform and optionally provide a change-of-basis for parametric patch evaluation in the tessellation evaluation shader. A set of level-of-detail parameters must also be set which determine the level of triangulation to be performed. For triangular patches, 3 outer edge level-of-detail parameters and 1 inner edge level-of-detail parameter must be set. The level-of-detail parameters are then passed into a fixed-function tessellator unit which will subdivide the patch and produce a set of triangles. Intuitively, larger level-of-detail parameters will produce a finer tessellation than smaller ones. Optionally, the level-of-detail values may be fractional which would allow for smooth adaptive tessellation.

2. Tessellation Evaluation Shader

The tessellation evaluation shader takes as input the transformed patch vertices from the tessellation control shader. The execution of the evaluation shader is many-to-few with respect to the control shader since the tessellation and generation of triangles takes place before the evaluation shader executes. The evaluation shader can see as input all vertices of the patch. Also as input, parametric variables are available to

the evaluation shader. The parametric variables are used to evaluate the patch and output the final position/attributes of the current triangle vertex being processed. For a triangular patch, 3 variables u, v, and w are available, and for a quadrangular patch, 2 variables u and v are available. It is intuitive to think of these variables as linear interpolation variables. The simplest form of patch evaluation is linear interpolation in which the corners of the patch are used in combination with the parametric variables to produce a bi-linearly or tri-linearly interpolated surface of points. Various options are available to the tessellation evaluation shader as well, such as how the primitives being received by the shader are ordered (counter-clockwise or clockwise).

3. Using Tessellation Control and Evaluation Shaders

For this research, the tessellation functionality of newer GPUs was used in order to alleviate the linear nature of tetrahedral elements. The tetrahedral boundaries are drawn using triangle patches consisting of 3 vertices. The tessellation shaders are used to trilinearly interpolate along the surfaces of the patches, producing a new collection of triangles. During the evaluation stage, the displacement texture is sampled at the given texture coordinate and the positions are offset along the normal based on the displacement texture. It should be noted that this will cause surface interpenetration in the interior of the object being simulated, but these artifacts are hardly seen at all during fracture. The purpose of using the tessellation shader is to allow individual pieces to not appear so lienar in nature. The surface of a tetrahedron is only displaced if that tetrahedral wall is completely marked as inside the object. Also, the amount of displacement must be small so that surfaces on the interior do not penetrate surfaces on the exterior of the object, producing visual artifacts even when no fracturing is being used. See Figure 18 for an example of how tessellation is being used. Figure 19 also shows how adpative, view-dependent level of detail could be used in conjunction

with the tessellation shader.

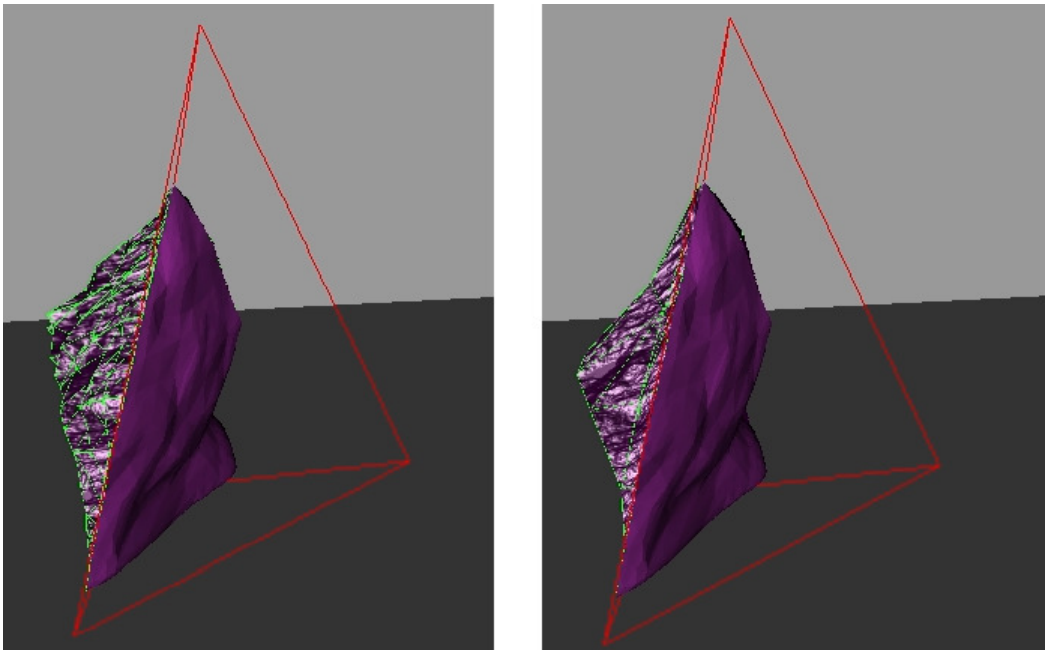


Fig. 18. A portion of the mesh which is interior is tessellated to provide an increase in geometry. Node the extrusion is exagerated from what would be used in reality. Left: Detailed geometry. Right: Coarser geometry of the same patch.

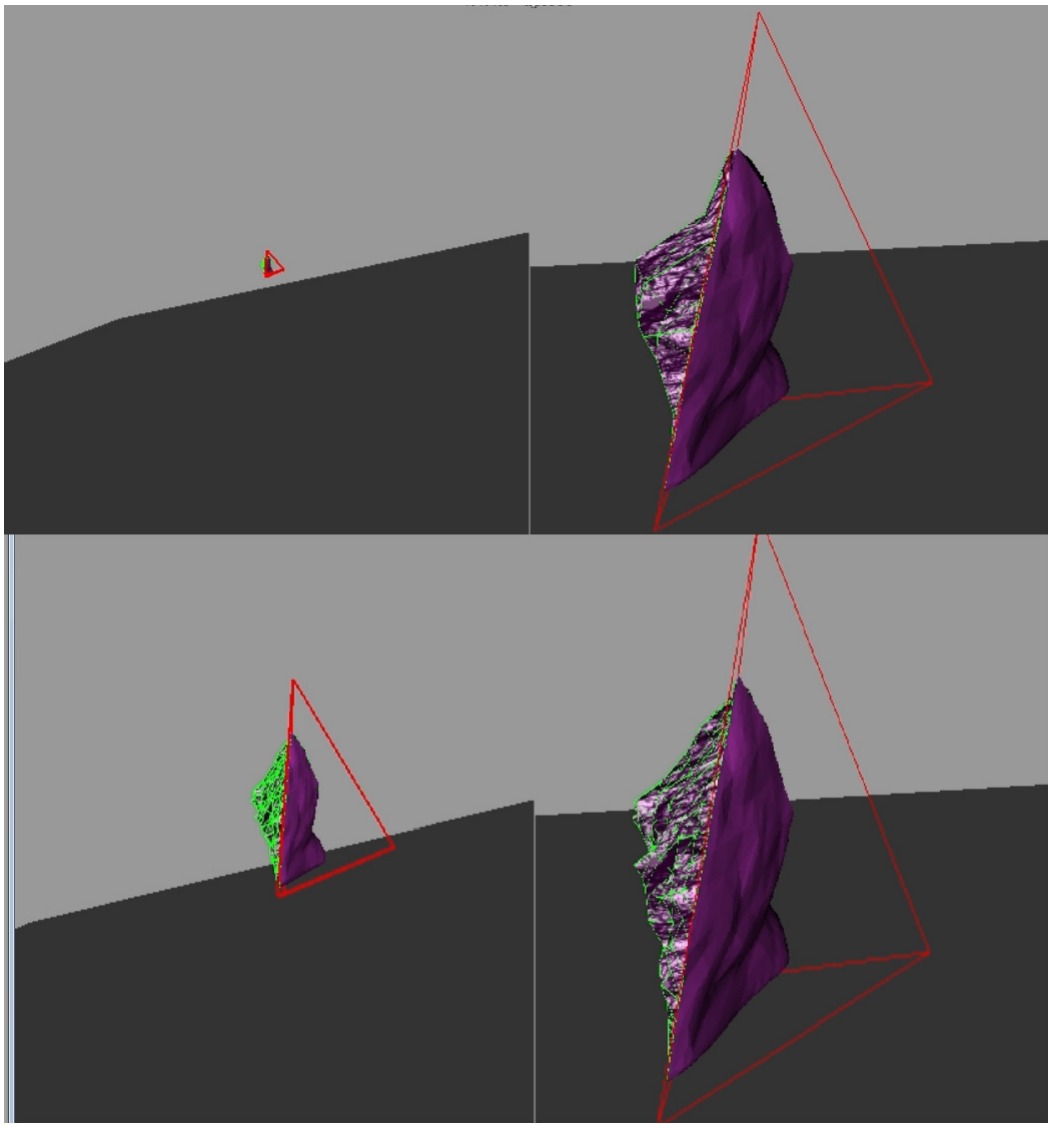


Fig. 19. Top: Far away from the geometry, the tessellation is allowed to be coarser. Bottom: Closer to the geometry will call for finer geometry. The images on the left represent the actual view the user would see, and the right side is a close-up of the actual tessellation being produced.

CHAPTER VI

RESULTS AND DISCUSSION

This chapter will present the results of the implementation of this research as well as a discussion on the various advantages and disadvantages, possible improvements, and future research. Additionally, an experimental analysis of the error associated with the force mapping involved in collision detection is included.

A. Setup

1. Computer Used

All tests were run on an Intel i5-2500K CPU running Windows 7 64-bit with 8 GB of RAM. The graphics cards used were two Nvidia GTX 460s running in SLI. For the purposes of implementing and analyzing the algorithms used, multi-threading was not utilized in the implementation. Future research will include utilizing both CPU and GPU multi-threading in order to efficiently speed up the simulation.

2. Meshes

The two models which were used were the bunny and armadillo-man from the Stanford repository and subsequently simplified to various polygon counts. The control meshes for the models were designed first as a triangle mesh enclosing the graphical mesh. The triangle mesh was then tetrahedralized using TetGen [33]. The following chart lists the tetrahedral meshes used for each model as well as the tetrahedron count and display triangle count as a result. Please note that the increase in tetrahedron count increases the display triangle count as well due to boundaries.

B. Analysis

The preprocessing stage is responsible for generating the collision geometry and various textures for use during simulation. This section presents the meshes used in the simulation, experimental error analysis of the collision geometry, and timings of the preprocessing section as well as sample images from real simulations.

1. Collision Geometry

The relative error associated with mapping forces from the collision geometry to the tetrahedra was used to determine the visual accuracy of the simulation. Figure 20 lists the number of spheres that were actually added to the simulation as well as the largest error associated during collision time. The errors represent the amount of graphical triangles which actually inter-penetrate environmental surfaces.

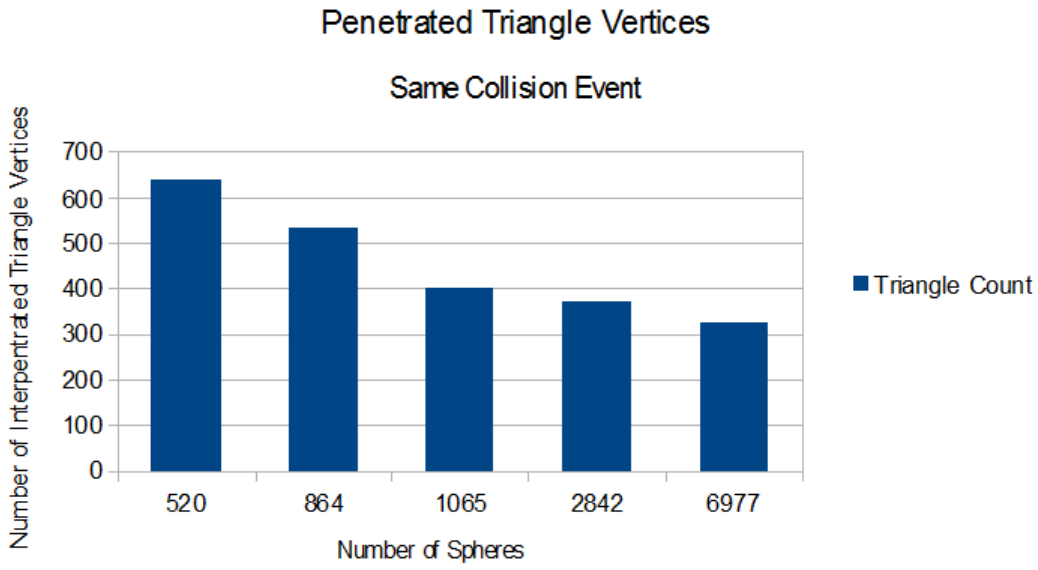


Fig. 20. Graph of number of vertices which are interpenetrating a surface with multiple counts of collision geometry.

As can be seen in Figure 20, as the number of spheres used in the collision detection increases, the amount of triangles which interpenetrate the surfaces is reduced. The reduction in error is intuitive because as more spheres are added (by making the resolution smaller and thus the radii of the spheres smaller), the more acurately the spheres comform to the actual geometry.

2. Timings

Since one of the goals was to decouple the graphical representation of the simulation from the physical representation, the timings related to each section are shown. What was being tested was whether the graphical display time went up as a result of increasing tetrahedron count, and if the tetrahedron count went up as a result of increase graphical display triangles. The results of the timings can be seen in Figure 21 and in Figure 22 for the Stanford Bunny model.

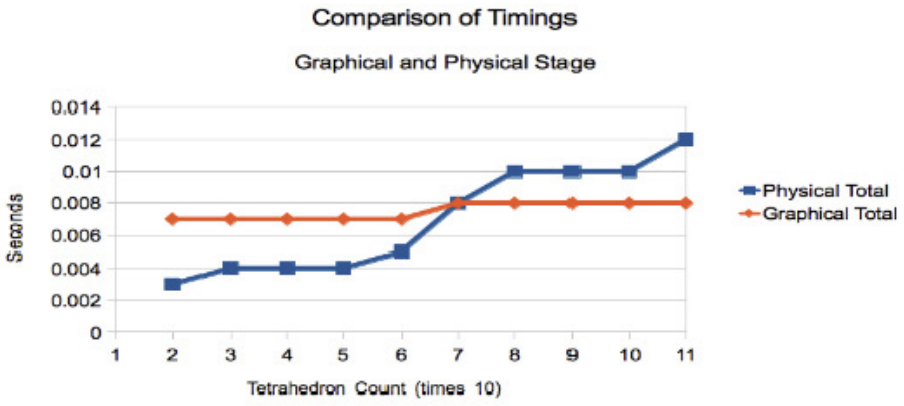


Fig. 21. Timings for physical and graphical simulation comopnents. As the tetrahedron count increases, the display simulation time does not increase substantially.

As can be seen in Figure 21, as the tetrahedron count gets higher, only the physical portion is substantially affected. Most of the time spent in the physical update was in the linear system solver when performing the implicit integration step. The

replication will increase the graphical timing a bit (as can be seen when the tetrahedron count reaches around 70), but is nothing compared to the amount of increase in the physical update step. A consequence of using the simple spherical collision detection scheme is that an increase in tetrahedra does not necessarily increase the time to perform collision detection substantially. One reason for this is because the spheres are mostly the same volume which means that there will be a similar count of total collision spheres given a high tetrahedron count compared to a low tetrahedron count. If more substantial geometry were to be used for collision detection such as a coarse triangle mesh, there may be replication along the boundary which would increase the amount of collision geometry with an increase in tetrahedron count. By using the spherical collision system, the collision geometry complexity is decoupled from the complexity of the tetrahedron mesh. Unfortunately, an increase in tetrahedron count will cause input display triangles to be replicated along the boundaries of the tetrahedra. The time to update the tetrahedral vertices on the GPU was not taken into account in these timings but was instead included in the graphical timing results which could also be a reason for the slight increase in display time. Finally, the graphics could be adversly affected since with more tetrahedra there will be more painted walls to draw for the user. In practice this has presented only a miniscule increase in display time. Note that no parallization optimization have been performed.

As can be seen in Figure 22, as the triangle count of the simulation is allowed to increase, the time to perform the graphical portion of the simulation increases substantially. The physical update time stays relatively the same while the triangle count rises. This result is to be expected since the physical step is completely decoupled from the graphical step. The graphical results are using a mesh consisting of only 10 tetrahedra.

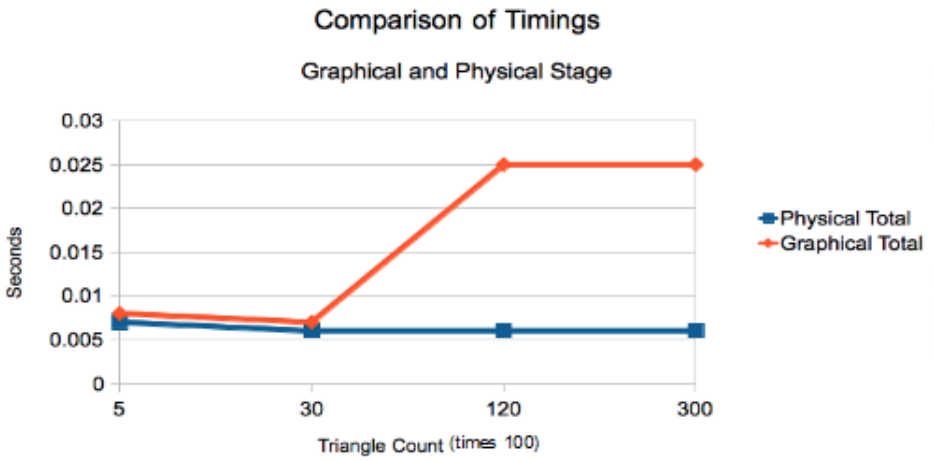


Fig. 22. Timings for physical and graphical simulation components. As the triangle count increases, the physical update step stays relatively the same which means the display and physical portions are adequately decoupled.

3. Texturing

During the preprocessing phase the walls of the tetrahedra are painted with the texture that is used to display the interior portions of the mesh (see Figure 23).

Point-wise sampling is performed during texture look-up instead of linear interpolation or mip-mapping in order to prevent portions of one face leaking into the other face of the tetrahedron. Point-side sampling will cause noticeable gaps along the boundary, but as long as the resolution of the textures is high enough then the effects are not noticeable unless zooming into the geometry very closely. The gaps appear because the square texels associated with the texture can not perfectly conform to the linear boundary of the model. Future research in this area will include using various types of texturing schemes in order to more accurately fit the texture to the contour lines of the face.

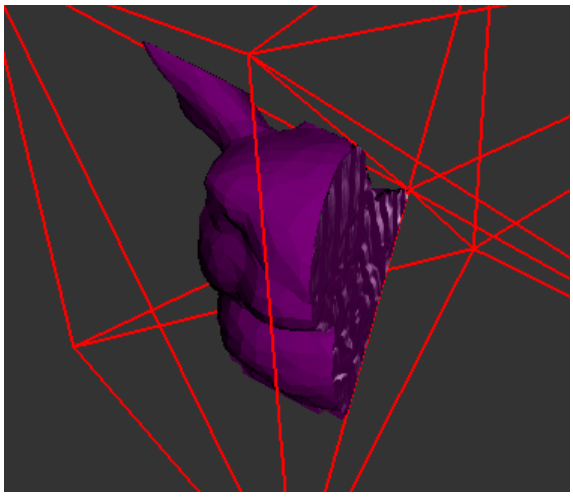


Fig. 23. The bunny's face intersects with the tetrahedral boundary. The painted texture can clearly be seen in the wavy portion. Note the specular high-lights.

4. Display Triangle Count

Unfortunately, the addition of graphical triangles leads to an increase in the amount of memory consumption by the algorithm, but usually the tetrahedral mesh is very coarse and the increase is not substantial in practice. Furthermore, since the point of this research is to focus on a simplification of the collision detection, the mesh should remain as simple as possible (many orders of magnitude less than the graphical mesh).

Note that the addition of triangles is not a form of remeshing in the typical sense. All triangles are added during preprocessing which prevent us from having to remesh on-the-fly during fracture simulation. The results of the display triangle count can be seen in Table II where the columns represent the number of graphical triangles which were replicated 1 through 7 times. As the tetrahedron count goes up, the number of graphical triangles which must be replicated 2 or more times goes up as well due to more tetrahedral edge crossings. Even with a large number of tetrahedra, the total count of the graphical triangles rises to only about 150% of the original.

Table II. Counts for replicated triangles

Tetrahedron Count	1X	2X	3X	4X	5X	6X	7X	Total
10	4315	661	18	6	0	0	0	5715
20	3871	1049	62	14	4	0	0	6231
31	3575	1275	109	32	5	4	0	6629
41	3353	1467	137	29	6	6	0	6880
50	3358	1445	144	37	7	5	2	6907
60	3258	1532	145	39	14	10	2	7057
72	3040	1713	174	42	16	13	2	7328
82	2972	1733	204	55	20	10	6	7472
91	2972	1733	204	55	20	10	6	7472
100	2856	1848	209	51	16	17	3	7586

Figure 24 shows the model with different colors corresponding to regions which had triangles duplicated.

5. Early Results

Early results were obtained by using a simplified collision detection model and no displacement mapping. The collision detection occurred completely on the tetrahedral boundaries. The early implementation was designed to test the embedding and display algorithm. Initially, a procedural control mesh was used which was created by generating a uniform grid of cubes around the volume of the solid object. The grid was then subdivided into tetrahedra. One reason for using a procedural control mesh in the beginning was to see how the number of rendered triangles would compare to the original mesh's triangle count when the triangle count increased. Triangles which cross tetrahedral boundaries will be stored once for each tetrahedron. Unfortunately, with a large amount of tetrahedra the number of displayed triangles could actually be double the amount of original triangles. In order to prevent a large number of triangles from being created due to boundaries, the initial tetrahedral control mesh must be small. The early display triangle counts can be seen in figure 5.

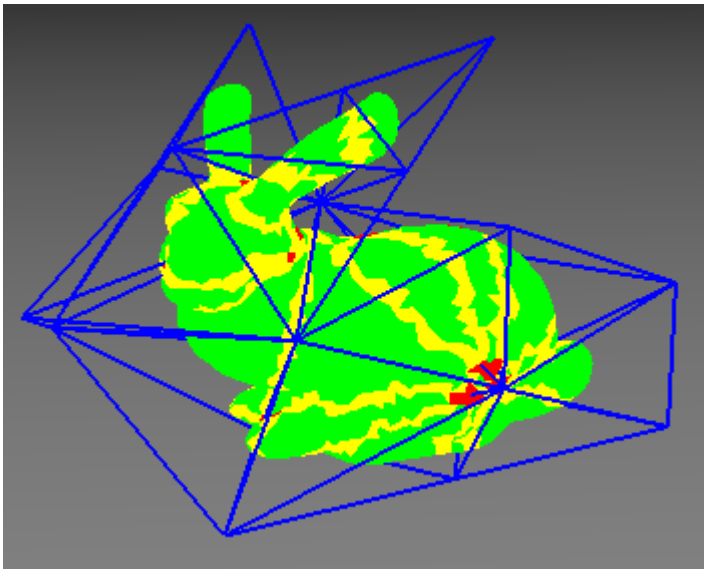


Fig. 24. Same regions of the model with a different resolution of tetrahedral mesh. A triangle that is not duplicated is shown in green, a triangle that is duplicated only once is yellow, and a triangle that is duplicated 2 or more times is shown in red.

As can be seen in Figure 25, the control mesh follows a uniform grid alignment. The tetrahedral boundaries do not align up with the underlying graphical mesh. Current research uses an artist rendered physical mesh in order to more accurately align the control mesh with the graphical mesh.

The collision detection scheme was originally tested by using the tetrahedra as the bounding geometry (see Figure 26). Obviously artifacts resulted from treating empty regions of space as filled.

One noticeable difference between earlier results using a uniform grid and results using a control mesh is that the number of graphical triangles which are replicated is significantly more in the case of using a uniform grid (see Figure 27). The reason for this difference is that the early results used a regular grid and subdivided to obtain the tetrahedra. The newer results use tetrahedra formed from a control mesh which

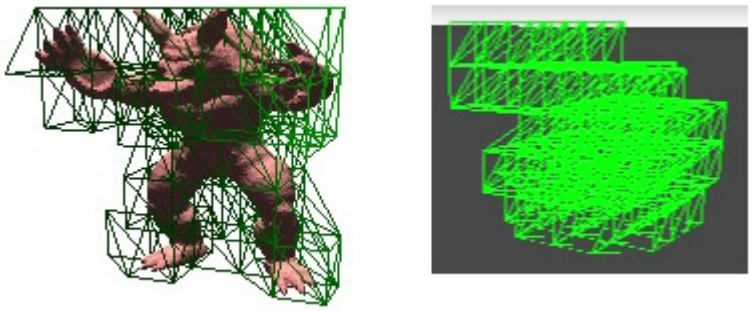


Fig. 25. Different control meshes used in early experiments. Left: Armadilloman with control cage. Right: Control cage of bunny.

Fig. 26. Early results using the entire tetrahedron as collision geometry. A ball hits the armadilloman from the left of the screen.

more accurately corresponds to the geometry and may have subdivisions in places such as wrists or necks which are thinner and have fewer triangles. The comparison results of the graphical triangle replication count shows that it is important to use a control mesh which accurately depicts its embedded geometry.

C. Discussion

This section will present some discussion of the algorithm that was developed. Pitfalls that were encountered in the process of designing the algorithm will be discussed as well as well as possible future work.

Bunny # Original Triangles	Subdivision	# Tetrahedra	# Rendered Triangles
5k	3	48	11296
5k	6	406	17156
40k	3	48	59192
40k	6	421	88460

Armadillo-Man # Original Triangles	Subdivision	# Tetrahedra	# Rendered Triangles
5k	3	48	10432
5k	6	249	16204
20k	3	48	31768
20k	6	266	47984

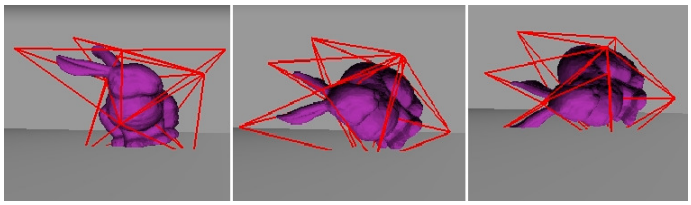
Fig. 27. Early display triangle counts for various tetrahedron counts.

1. Collision Geometry

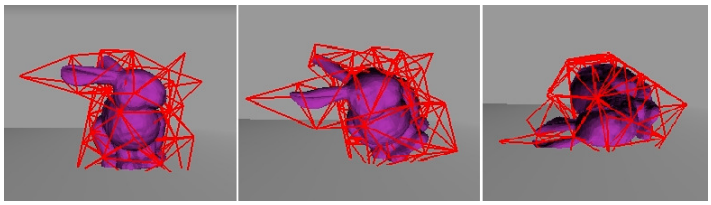
The spherical-based collision geometry provided a simple method to perform collision detection with the object without using the entire surface geometry. The simplicity detection algorithm also results in various artifacts. One obvious artifact is that the graphical mesh's geometry can interpenetrate in regions where a sphere might not be present. Adding more spheres will help to alleviate this issue but it will also increase the computational time of the simulation. This type of collision detection could be improved by using geometry which more closely aligns with the graphical mesh such as a simplified collision mesh. The generation of such a mesh is difficult to perform automatically (which is desired from a production pipeline point-of-view). Future research in this area would include how to automatically generate a collision mesh which would reduce the artifacts associated with the sphere-based approach while being able to be generated automatically during preprocessing.

2. Choosing Proper Material Parameters

The choice of material values is important in the simulation since the underlying equations are derived from continuum mechanics for real materials. Various parameters were experimented with for this research in order to produce different results. Many texts on the subject of continuum mechanics and fracture analysis include tables of material parameters from real items which were empirically measured. In our case, the user interface of the simulation contains slots for manipulating and experimenting with different material parameters. See Figure 28 for an example of how material parameters and control mesh complexity can change the way the mesh interacts with the environment.



(a) Stiff Mesh



(b) Non-stiff

Fig. 28. An example of different material parameters and different resolution tetrahedral meshes. 28(a) A stiff mesh. 28(b) A mesh which deforms more due to less stiff material parameters and more DoF due to more tetrahedra.

It can be seen from and Figure 29 and Figure 30 that using normal mapping along with displacement mapping greatly enhances the linear boundaries of the physical mesh. One issue with using displacement mapping is that the interior may interpenetrate outer geometry if the tetrahedral face is too close to the surface. This issue can be greatly alleviated by displacing the geometry with a maximum displacement factor of the distance to the exterior polygons. Unfortunately this information is not available for the GPU and involves computing a distance field for ever texel sample of every tetrahedral face. Instead of computing the distance field, which is both computationally and memory intensive, we only displace the geometry for tetrahedral faces which are contained completely inside the mesh and use normal mapping for the other faces. While interpenetration artifacts may still occur if the displacement is too large, tetrahedral faces which are completely inside the mesh are usually far enough away from the exterior graphical geometry. Furthermore, allowing these completely interior tetrahedral faces to only displace inwards is another option. Both options were tested for this research.

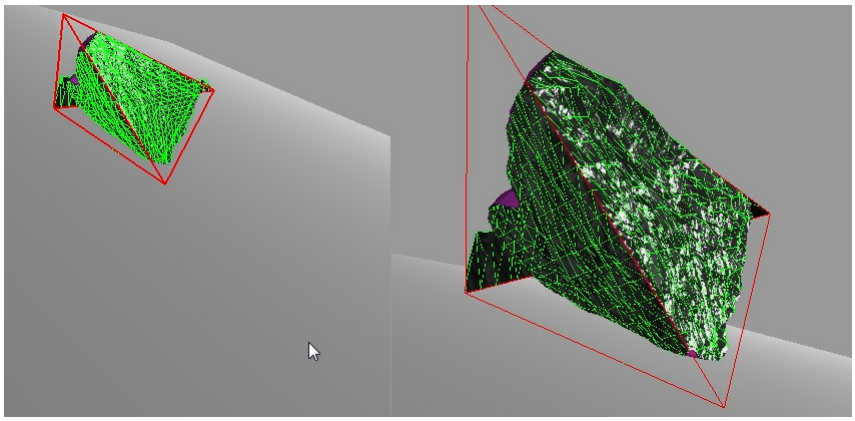


Fig. 29. Highly tessellated subregion. Left: Real distance from camera. Right: Close-up.

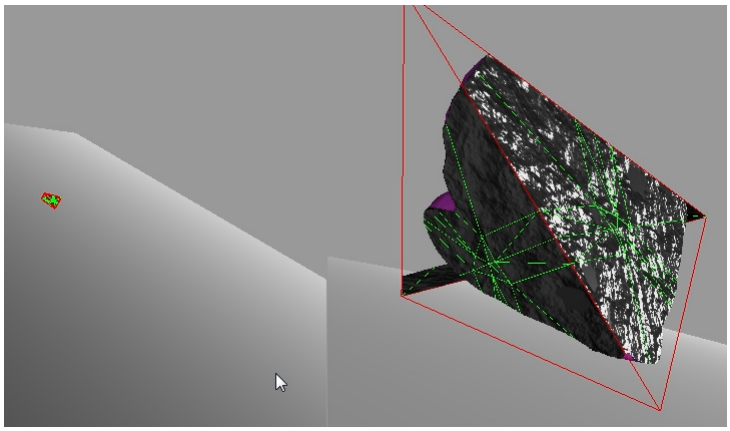


Fig. 30. Low tessellation due to camera being far away. Bump mapping is not affected. Left: Real distance from camera. Right: Close-up.

The use of GPU tessellation techniques is also restricted to newer GPU hardware. For older hardware which does not implement the tessellation shader functionality, using normal mapping would be a fall-back for displaying the interior portions of the tetrahedra.

CHAPTER VII

CONCLUSION

The method that has been developed allows for an efficient means to simulate embedded geometry involving fracture without a need to remesh the graphical geometry during the simulation. Various GPU techniques were used to both display the graphical mesh and display portions of the physical mesh which may be within the solid object without the need to compute and store extra triangles along those faces. The algorithm requires a semi-expensive preprocessing step in order to create the assets, but it allows for a complete separation between the graphical mesh and the physical mesh. The regions of space which are within the intersections of the graphical mesh and tetrahedral mesh are textured in order to simulate a solid region during fracture. Future research will focus on adding parallelism to the simulation, creating algorithms to automatically generate the control mesh and boundary textures, use of more exact algorithms for displaying the boundary tetures, and continuing research into image based collision techniques to handle fractured geometry.

A. Future Research

1. Utilizing Parallelism

The fundamentally most time consuming process in the physical loop is time-stepping. While using implicit time-stepping allows for much larger time steps, there are various portions which can be trivially parallelized. Since the main focus of this thesis has been on designing and testing the embedding algorithm, parallelization has not been utilized. The physical calculations of stress and strain on an element are fundamentally independent operations. One area of possible parallelization is during

stress/strain calculations since the only coordination between the elements comes into play during stiffness matrix assembly and time integration.

2. Multiple Materials in the Same Tetrahedral Element

It may be that multiple materials could be embedded elements. Many researchers have simply created a new element for each set of connected material within the object (see Nesme et al. [14]). This formulation presents challenges to fracture since the nodes between adjoining tetrahedra are connected.

3. Image-based Collision Techniques

Image based collision techniques still offer the promise of accurate and robust collision detection in screen space which is beneficial for our algorithm since portions of the mesh (tetrahedral faces) do not have an explicit representation. Furthermore, the simplified spherical collision techniques presented in this research allow for interpenetration artifacts in a virtual world. Future research into image-based collision techniques will focus on reducing the CPU to GPU data transfer as well as the number of rendering passes required. The resolution of the image-based collision technique as pieces of the mesh break apart and become further away must also not restrict and diminish the quality of the physical simulation.

REFERENCES

[1] T.W. Sederberg and S.R. Parry, "Free-form deformation of solid geometric models," *ACM SIGGRAPH Computer Graphics*, vol. 20, no. 4, pp. 151–160, 1986.

[2] J.F. O'Brien and J.K. Hodgins, "Graphical modeling and animation of brittle fracture," in *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*. ACM Press/Addison-Wesley Publishing Co., 1999, pp. 137–146.

[3] E.G. Parker and J.F. O'Brien, "Real-time deformation and fracture in a game environment," in *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. ACM, 2009, pp. 165–175.

[4] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer, "Elastically deformable models," in *ACM SIGGRAPH Computer Graphics*. ACM, 1987, vol. 21, pp. 205–214.

[5] D. Terzopoulos and K. Fleischer, "Modeling inelastic deformation: viscolelasticity, plasticity, fracture," in *ACM SIGGRAPH Computer Graphics*. ACM, 1988, vol. 22, pp. 269–278.

[6] D. Baraff and A. Witkin, "Large steps in cloth simulation," in *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*. ACM, 1998, pp. 43–54.

[7] J.F. O'Brien, A.W. Bargteil, and J.K. Hodgins, "Graphical modeling and animation of ductile fracture," in *ACM Transactions on Graphics (TOG)*. ACM, 2002, vol. 21, pp. 291–294.

[8] M. Hauth, O. Etzmuß, and W. Straßer, "Analysis of numerical methods for the simulation of deformable models," *The Visual Computer*, vol. 19, no. 7, pp. 581–600, 2003.

[9] P. Volino and N. Magnenat-Thalmann, "Comparing efficiency of integration methods for cloth simulation," in *Computer Graphics International 2001. Proceedings*. IEEE, 2001, pp. 265–272.

[10] S. Coquillart, "Extended free-form deformation: a sculpturing tool for 3d geometric modeling," *ACM SIGGRAPH Computer Graphics*, vol. 24, no. 4, pp. 187–196, 1990.

[11] R. MacCracken and K.I. Joy, "Free-form deformations with lattices of arbitrary topology," in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. ACM, 1996, pp. 181–188.

[12] P. Faloutsos, M. Van de Panne, and D. Terzopoulos, "Dynamic free-form deformations for animation synthesis," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 3, no. 3, pp. 201–214, 1997.

[13] Z. Melek and J. Keyser, "Driving object deformations from internal physical processes," in *Proceedings of the 2007 ACM Symposium on Solid and Physical Modeling*. ACM, 2007, pp. 51–59.

[14] M. Nesme, P.G. Kry, L. Jeřábková, and F. Faure, "Preserving topology and elasticity for embedded deformable models," in *ACM Transactions on Graphics (TOG)*. ACM, 2009, vol. 28, p. 52.

[15] J. Smith, A. Witkin, and D. Baraff, "Fast and controllable simulation of the

shattering of brittle objects," in *Computer Graphics Forum*. Wiley Online Library, 2001, vol. 20, pp. 81–91.

[16] M. Müller, L. McMillan, J. Dorsey, and R. Jagnow, "Real-time simulation of deformation and fracture of stiff materials," *Computer Animation and Simulation 2001*, pp. 113–124, 2001.

[17] M. Müller, J. Dorsey, L. McMillan, R. Jagnow, and B. Cutler, "Stable real-time deformations," in *Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, July*, 2002, pp. 21–22.

[18] O. Etzmuß, M. Keckeisen, and W. Straßer, "A fast finite element solution for cloth modelling," in *Computer Graphics and Applications, 2003. Proceedings. 11th Pacific Conference on*. IEEE, 2003, pp. 244–251.

[19] M. Müller and M. Gross, "Interactive virtual materials," in *Proceedings of Graphics Interface 2004*. Canadian Human-Computer Communications Society, 2004, pp. 239–246.

[20] M. Muller, M. Teschner, and M. Gross, "Physically-based simulation of objects represented by surface meshes," in *Computer Graphics International, 2004. Proceedings*. IEEE, 2004, pp. 26–33.

[21] Z. Bao, J.M. Hong, J. Teran, and R. Fedkiw, "Fracturing rigid materials," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 13, no. 2, pp. 370–378, 2007.

[22] N. Molino, Z. Bao, and R. Fedkiw, "A virtual node algorithm for changing mesh topology during simulation," in *ACM SIGGRAPH 2005 Courses*. ACM, 2005, p. 4.

[23] Martin Wicke, Daniel Ritchie, Bryan Matthew Klingner, Sebastian Burke, Jonathan Richard Shewchuk, and James F. O'Brien, "Dynamic local remeshing for elastoplastic simulation," *ACM Trans. Graph.*, vol. 29, no. 4, pp. 49:1 – 49:11, 2010.

[24] E. Sifakis, K.G. Der, and R. Fedkiw, "Arbitrary cutting of deformable tetrahedralized objects," in *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. Eurographics Association, 2007, pp. 73–80.

[25] T.L. Anderson, *Fracture mechanics: fundamentals and applications*, CRC Pr I Llc, 1995.

[26] M. Pauly, R. Keiser, B. Adams, P. Dutré, M. Gross, and L.J. Guibas, "Meshless animation of fracturing solids," in *ACM Transactions on Graphics (TOG)*. ACM, 2005, vol. 24, pp. 957–964.

[27] M. Müller, B. Heidelberger, M. Teschner, and M. Gross, "Meshless deformations based on shape matching," in *ACM Transactions on Graphics (TOG)*. ACM, 2005, vol. 24, pp. 471–478.

[28] E. Sifakis, T. Shinar, G. Irving, and R. Fedkiw, "Hybrid simulation of deformable solids," in *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. Eurographics Association, 2007, pp. 81–90.

[29] C. Mendoza and C. Laugier, "Simulating soft tissue cutting using finite element models," in *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*. IEEE, 2003, vol. 1, pp. 1109–1114.

[30] J. Teran, S. Blemker, V. Hing, and R. Fedkiw, "Finite volume methods for the simulation of skeletal muscle," in *Proceedings of the 2003 ACM SIG-*

GRAPH/Eurographics Symposium on Computer Animation. Eurographics Association, 2003, pp. 68–74.

[31] J. Teran, E. Sifakis, S.S. Blemker, V. Ng-Thow-Hing, C. Lau, and R. Fedkiw, "Creating and simulating skeletal muscle from the visible human data set," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 11, no. 3, pp. 317–328, 2005.

[32] A. Nealen, M. Müller, R. Keiser, E. Boxerman, and M. Carlson, "Physically based deformable models in computer graphics," in *Computer Graphics Forum*. Wiley Online Library, 2006, vol. 25, pp. 809–836.

[33] Hang Si, "Tetgen: A quality tetrahedral mesh generator and 3d delaunay triangulator.," Feb. 2012.

[34] G.E. Farin, *Curves and surfaces for CAGD: a practical guide*, Morgan Kaufmann Pub, 2002.

[35] F. Faure, S. Barbier, J. Allard, and F. Falipou, "Image-based collision detection and response between arbitrary volume objects," in *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. Eurographics Association, 2008, pp. 155–162.

[36] J. Shade, S. Gortler, L. He, and R. Szeliski, "Layered depth images," in *Proceedings of the 25th annual Conference on Computer Graphics and Interactive Techniques*. ACM, 1998, pp. 231–242.

[37] C. Everitt, "Interactive order-independent transparency," *White paper, nVIDIA*, vol. 2, no. 6, pp. 7, 2001.

[38] A. Witkin and D. Baraff, "Physically based modeling," Feb. 2012.

[39] J.D. Foley, *Computer graphics: principles and practice*, Addison-Wesley Professional, 1996.

[40] L. Szirmay-Kalos and T. Umenhoffer, "Displacement mapping on the gpustate of the art," in *Computer Graphics Forum*. Wiley Online Library, 2008, vol. 27, pp. 1567–1592.

VITA

Name:	Billy Russell Clack
Address:	Department of Mathematics, Mailstop 3368, Texas A&M University, College Station, TX 77843-3368 Rm. 608 D
Email Address:	scyfris@gmail.com
Education:	B.M., Music, Stephen F. Austin State University, 2009 M.S., Computer Science, Texas A&M University, 2012